

# APL IS EASY!

Jerry R. Turner  
STSC, Inc.

John Wiley & Sons, Inc.  
New York • Chichester • Brisbane • Toronto • Singapore

This publication is designed to provide accurate and authoritative information in regard to the subject matter covered. It is sold with the understanding that the publisher is not engaged in rendering legal, accounting, or other professional service. If legal advice or other expert assistance is required, the services of a competent professional person should be sought. FROM A DECLARATION OF PRINCIPLES JOINTLY ADOPTED BY A COMMITTEE OF THE AMERICAN BAR ASSOCIATION AND A COMMITTEE OF PUBLISHERS.

Copyright © 1987 STSC, Inc.

All rights reserved. Published simultaneously in Canada.

Reproduction or translation of any part of this work beyond that permitted by section 107 or 108 of the 1976 United States Copyright Act without the permission of the copyright owner is unlawful. Requests for permission or further information should be addressed to the Permission Department, John Wiley & Sons, Inc.

**Library of Congress Cataloging in Publication Data**

APL Is Easy!

ISBN 0-471-85882-X

Printed in the United States of America

87 88 10 9 8 7 6 5 4 3 2 1



# Contents

---

|                      |                                                                  |    |
|----------------------|------------------------------------------------------------------|----|
| <b>Introduction</b>  | <b>Important Information about This Book</b>                     |    |
|                      | Why Read This Book?                                              | 1  |
|                      | Who Should Read This Book?                                       | 2  |
|                      | How Is the Book Organized?                                       | 2  |
|                      | What Is the Book's Format?                                       | 3  |
|                      | Where Do I Go from Here?                                         | 4  |
|                      | Your Comments Are Welcome                                        | 4  |
| <br><b>Chapter 1</b> | <br><b>What Is a Program?</b>                                    |    |
|                      | Programs and Data                                                | 5  |
|                      | Loading a Workspace                                              | 6  |
|                      | Running a Program                                                | 7  |
|                      | Listing the Names of Programs                                    | 9  |
|                      | Listing the Names of Variables                                   | 9  |
|                      | Summary                                                          | 10 |
| <br><b>Chapter 2</b> | <br><b>Simple Arithmetic, Error Correction, and Data Storage</b> |    |
|                      | Using APL Like a Desk Calculator                                 | 11 |
|                      | Handling Lists of Numbers                                        | 12 |
|                      | Negative Numbers                                                 | 13 |
|                      | Order of Execution                                               | 14 |
|                      | Right-to-Left Execution                                          | 14 |
|                      | Use of Parentheses                                               | 15 |
|                      | Error Correction                                                 | 15 |
|                      | Interrupting Input                                               | 16 |
|                      | Untype                                                           | 16 |
|                      | Delete                                                           | 16 |
|                      | Storing Data in Variables                                        | 16 |
|                      | Shape Function                                                   | 18 |
|                      | Catenate Function                                                | 18 |
|                      | Working with Tables of Numbers                                   | 19 |
|                      | Reshape Function                                                 | 19 |
|                      | Arithmetic with Tables                                           | 21 |

|                  |                                                                |    |
|------------------|----------------------------------------------------------------|----|
|                  | Function Symbols with Double Uses                              | 22 |
|                  | Arguments                                                      | 23 |
|                  | Monadic and Dyadic Functions                                   | 23 |
|                  | Summary                                                        | 24 |
|                  | Exercises                                                      | 25 |
| <b>Chapter 3</b> | <b>Character Data, More Arithmetic, and More APL Functions</b> |    |
|                  | Character Data                                                 | 28 |
|                  | Format Function                                                | 31 |
|                  | Character Matrices                                             | 31 |
|                  | Maximum                                                        | 32 |
|                  | Minimum                                                        | 34 |
|                  | Reduction                                                      | 34 |
|                  | Maximum Reduction                                              | 34 |
|                  | Plus Reduction                                                 | 35 |
|                  | Times Reduction                                                | 35 |
|                  | Minimum Reduction                                              | 35 |
|                  | Reduction with Matrices                                        | 35 |
|                  | Scan                                                           | 37 |
|                  | Indexing                                                       | 37 |
|                  | Using Index with Variables                                     | 38 |
|                  | Replacing Individual Elements                                  | 38 |
|                  | Indexing into Matrices                                         | 39 |
|                  | Indexing with Character Data                                   | 41 |
|                  | Count                                                          | 41 |
|                  | Roll                                                           | 42 |
|                  | Summary                                                        | 44 |
|                  | Exercises                                                      | 44 |
| <b>Chapter 4</b> | <b>Selecting and Analyzing Data</b>                            |    |
|                  | An Expression for Selecting Data                               | 50 |
|                  | Compression                                                    | 51 |
|                  | Relational Functions                                           | 52 |
|                  | Selecting Character Data                                       | 52 |
|                  | Counting Occurrences                                           | 53 |

|                              |    |
|------------------------------|----|
| Meeting Two Conditions       | 54 |
| And                          | 54 |
| Or                           | 54 |
| Checking All of the Data     | 55 |
| And Reduction                | 55 |
| Or Reduction                 | 56 |
| Selecting Data from Matrices | 56 |
| Summary                      | 60 |
| Exercises                    | 61 |

## Chapter 5

|                                          |    |
|------------------------------------------|----|
| <b>Writing Programs in APL</b>           |    |
| Why Write Programs?                      | 65 |
| An Example of an APL Program             | 65 |
| Defining a Function                      | 66 |
| Executing a Defined Function             | 67 |
| Correcting Mistakes                      | 68 |
| Replacing an Entire Line                 | 68 |
| Correcting Part of a Line                | 69 |
| Displaying a Function Definition         | 70 |
| Inserting a Line                         | 72 |
| Erasing a Line                           | 73 |
| Summary of Editing Commands              | 74 |
| Adding Character Data to Functions       | 75 |
| Different Function Forms                 | 76 |
| Functions with One Argument              | 76 |
| Functions with Two Arguments             | 77 |
| Functions with No Arguments              | 78 |
| Explicit Results                         | 79 |
| Defining Functions with Explicit Results | 80 |
| Function Syntax                          | 82 |
| Summary                                  | 83 |
| Exercises                                | 83 |

## Chapter 6

|                          |    |
|--------------------------|----|
| <b>Formatting</b>        |    |
| What Is Formatting?      | 91 |
| Dyadic Format            | 91 |
| Format Pairs             | 91 |
| Formatting Whole Numbers | 91 |

---

|                  |                                          |     |
|------------------|------------------------------------------|-----|
|                  | Formatting Numbers with Decimal Places   | 92  |
|                  | Formatting Columns Differently           | 92  |
|                  | Formatting a Matrix                      | 92  |
|                  | <code>⎕FMT</code>                        | 93  |
|                  | Format Phrases                           | 93  |
|                  | Decorations                              | 95  |
|                  | Repetition Factor                        | 97  |
|                  | Formatting Multiple Pieces of Data       | 97  |
|                  | Basic Reporting Functions                | 98  |
|                  | Summary                                  | 99  |
|                  | Exercises                                | 101 |
| <br>             |                                          |     |
| <b>Chapter 7</b> | <b>Storage Facilities: Workspaces</b>    |     |
|                  | What Is a Workspace?                     | 104 |
|                  | System Commands                          | 105 |
|                  | <code>⋄SAVE</code>                       | 105 |
|                  | <code>⋄CLEAR</code>                      | 106 |
|                  | <code>⋄FNS</code>                        | 107 |
|                  | <code>⋄VARS</code>                       | 107 |
|                  | <code>⋄ERASE</code>                      | 108 |
|                  | <code>⋄WSLIB</code> or <code>⋄LIB</code> | 108 |
|                  | <code>⋄WSID</code>                       | 108 |
|                  | <code>⋄DROP</code>                       | 109 |
|                  | <code>⋄COPY</code>                       | 110 |
|                  | <code>⋄OFF</code>                        | 111 |
|                  | System Functions                         | 111 |
|                  | Summary                                  | 111 |
| <br>             |                                          |     |
| <b>Chapter 8</b> | <b>Storage Facilities: Files</b>         |     |
|                  | What Is a File?                          | 113 |
|                  | Advantages of Using Files                | 113 |
|                  | Characteristics of Files                 | 114 |
|                  | File Functions                           | 114 |
|                  | <code>⎕FCREATE</code>                    | 115 |
|                  | <code>⎕FAPPEND</code>                    | 115 |
|                  | <code>⎕FSIZE</code>                      | 116 |
|                  | <code>⎕FREAD</code>                      | 117 |

---

|                   |                                   |     |
|-------------------|-----------------------------------|-----|
|                   | □FREPLACE                         | 118 |
|                   | □FUNTIE                           | 118 |
|                   | □FTIE                             | 119 |
|                   | □FNAMES                           | 119 |
|                   | □FNUMS                            | 119 |
|                   | □FERASE                           | 120 |
|                   | □FLIB                             | 120 |
|                   | Summary                           | 121 |
|                   | Exercises                         | 121 |
| <b>Chapter 9</b>  | <b>APL Programming Techniques</b> |     |
|                   | Interactive Functions             | 129 |
|                   | Evaluated Input                   | 129 |
|                   | Character Input                   | 130 |
|                   | Catenation with Matrices          | 131 |
|                   | Take                              | 132 |
|                   | Looping                           | 133 |
|                   | Tests for Conditional Branches    | 134 |
|                   | Line Labels                       | 135 |
|                   | Programming Aids                  | 137 |
|                   | Comments                          | 137 |
|                   | Local and Global Variables        | 138 |
|                   | Functions and Files               | 140 |
|                   | Summary                           | 148 |
|                   | Exercises                         | 149 |
| <b>Chapter 10</b> | <b>Exploring APL</b>              |     |
|                   | Sorting                           | 156 |
|                   | Rounding                          | 159 |
|                   | Locating Data                     | 159 |
|                   | Membership                        | 160 |
|                   | Where                             | 160 |
|                   | Table Lookup                      | 161 |
|                   | Converting Characters to Numbers  | 162 |
|                   | Scalar, Vector, or Matrix?        | 164 |
|                   | Rank                              | 164 |
|                   | Multidimensional Objects          | 165 |

---

|                                |                                              |
|--------------------------------|----------------------------------------------|
| Manipulating Data              | 166                                          |
| Ravel                          | 166                                          |
| Rotate                         | 167                                          |
| Overturn                       | 168                                          |
| Transpose                      | 169                                          |
| Grabbing Pieces of Data        | 170                                          |
| Drop                           | 170                                          |
| Assorted Math Functions        | 171                                          |
| Exponentiation                 | 171                                          |
| Logarithms                     | 171                                          |
| Outer Product                  | 171                                          |
| Inner Product                  | 172                                          |
| Matrix Multiplication          | 172                                          |
| Matrix Division                | 173                                          |
| Least-Squares Approximation    | 174                                          |
| Matrix Inversion               | 175                                          |
| Combinations                   | 175                                          |
| Trigonometric Functions        | 175                                          |
| Some APL Idioms                | 176                                          |
| Numeric or Character?          | 176                                          |
| Branch to Next Line            | 177                                          |
| Finding Indexes                | 178                                          |
| Locating Data Based on a Test  | 178                                          |
| Replacing A with B in a Vector | 178                                          |
| Testing for an Empty Result    | 179                                          |
| Finding a Substring            | 179                                          |
| <b>Appendixes</b>              |                                              |
| A                              | Solutions to Exercises 181                   |
| B                              | Error Messages and What to Do about Them 208 |
| C                              | Keyboard Template 215                        |
| D                              | APL Programs Used in This Book 216           |
| <b>Glossary</b>                | 228                                          |
| <b>Index</b>                   | 235                                          |

---

## Introduction

### Important Information about This Book

---

#### **WHY READ THIS BOOK?**

APL is an extremely concise and powerful computer programming language. You can learn to do very interesting things with only a little knowledge of the language.

APL (A Programming Language) was originally conceived in the late 1950s by Kenneth E. Iverson, then a professor of mathematics at Harvard University. It was initially used as a mathematical notation and not as a computer programming language. In the mid-1960s, the notation was implemented as a programming language for use by IBM's central research staff at the company's Thomas J. Watson Research Laboratory. APL has now been implemented by companies worldwide on a wide variety of hardware and operating systems, from large mainframes to microcomputers.

STSC, Inc. was organized in 1969 to provide an improved interactive time-sharing service based on the APL programming language, and has since expanded to include a full range of APL products for mainframes, minicomputers, and microcomputers.

As APL developed, more and more users discovered that APL offered significant productivity advantages when compared with more traditional languages such as FORTRAN, COBOL, and PL/I. As the language became more popular, STSC recognized the need for an introductory tutorial to the language so that new computer users could learn the language easily. This book was written to meet that need. *APL Is Easy!* has been adapted to fit most commercial APL systems, including IBM's VS APL, I. P. Sharp Associates' SHARP APL, and STSC's APL\*PLUS Systems, and concentrates on elements of the language common to them. If, however, something does not produce a result you expected, or does not work as shown in the book, refer to your system documentation or an experienced user of your APL system.

This book will take you through your first 10 to 15 hours of learning APL and teach you to do some useful things with what you learn. By the end of Chapter 3, you'll be storing data in the computer and making the computer "talk to you." By the end of Chapter 4, you'll be using the computer as an "electronic checkbook" to analyze your bank transactions. By the end of Chapter 5, you'll be writing programs to create simple reports and produce bar charts.

### **WHO SHOULD READ THIS BOOK?**

You need not have any previous computer experience to read this book. You *should* know how to start up your system before you begin reading, however. To learn about signing on, either consult the documentation provided for your system or ask an experienced user.

The book assumes that:

- You are sitting in front of a computer that is turned on and has APL loaded.
- You are familiar with your keyboard.
- You are using the APL \* PLUS System for the PC and have the workspaces *DEMOAPL* and *LESSONS* available.

If you are using APL on a different system, be sure to read your system documentation so that you are familiar with your system's keyboard. We've provided a blank keyboard layout for you in Appendix C. Fill in the characters on this keyboard in the proper locations for your system and use it as a quick reference throughout your work with this book.

We have also provided the APL code for the functions in the *LESSONS* workspace in Appendix D. If your system does not have this workspace, you can type in the code (see Chapter 5) or have someone more experienced with APL do it for you.

### **HOW IS THE BOOK ORGANIZED?**

*APL Is Easy!* contains 10 chapters, several appendixes, a glossary, and an index. The chapters contain the following information:

- Chapter 1 introduces the concept of programs and data.
- Chapter 2 explains how to do simple arithmetic and store data using APL.
- Chapter 3 explains how words and sentences are handled in APL, and also introduces some more ways to do arithmetic.
- Chapter 4 explains some ways to analyze your data.



- Chapter 5 explains how to write programs and make corrections in them.
- Chapter 6 introduces formatting with the `format` function and the system function `⌵FMT`. `⌵FMT` is a technique that gives you more control over the way the computer prints out information.
- Chapters 7 and 8 deal with the two storage facilities of APL, workspaces and files.
- Chapter 9 provides some additional programming tools, including the use of “friendly” interactive programs.
- Chapter 10 deals with more ways to analyze and manipulate your data.

Appendix A following Chapter 10 may be the most important part of the book—it contains the answers to the exercises and explanations of some of the more subtle points. Appendix B contains some common APL error messages, their causes, and solutions. Appendix C is the blank keyboard mentioned earlier, and Appendix D contains the programs used in the book.

Speaking of exercises—it’s really important that you work through as many of the exercises as you can, since that’s the best way to learn APL. Don’t worry if you don’t answer all the problems correctly—that’s what Appendix A is there for.

## WHAT IS THE BOOK’S FORMAT?

Each chapter contains an introduction, a number of sections (some containing subsections), and a summary. Most chapters include exercises at the end.

Section titles, subsection titles, and keywords all appear in the margins. Section titles are printed in bold, uppercase letters (such as **USING APL LIKE A DESK CALCULATOR**).

Subsection titles are printed in bold, with initial capital letters, such as **Negative Numbers** or **Arithmetic with Tables**. Keywords are printed in bold, lowercase letters, such as **immediate execution mode**. These are included to help you find things you’re looking for in the text.

APL examples are printed using APL letters (italic capital letters plus special APL symbols). Each example includes two parts:

- what you type in
- what the system replies

In APL, the user's entries are indented six spaces, and the system's replies are printed at the left-hand margin. For example,

|            |                                            |
|------------|--------------------------------------------|
| $9 \div 3$ | (You type this and press RETURN or ENTER.) |
| 3          | (The system responds with this.)           |

The system always returns to the six-space indent to wait for you to enter something new.

To clarify whether it is you or the system printing something, the user entries are in **bold type**, as shown in the example above.

Throughout this book, when we say "enter," we mean for you to do two things: (1) type something, and (2) press RETURN or ENTER.

**WHERE DO  
I GO FROM  
HERE?**

When you finish this book, you'll probably want to go on to a more detailed book on APL. Gilman and Rose's *APL: An Interactive Approach* is a good one. Your system's reference or user's guide would also be a good place to learn about specific things like full-screen editing, files, formatting, and the keyboard layout.

**YOUR  
COMMENTS  
ARE  
WELCOME**

We're eager to hear your comments on *APL Is Easy!* For your convenience, we've included a Feedback Form and postage-paid mailer at the end of this manual. If the form is missing, please send your comments to:

STSC, Inc.  
2115 East Jefferson Street  
Rockville, Maryland 20852

Be sure to include your name and address and the name of the manual.

## Chapter 1

# What Is a Program?

---

This chapter explains what a computer program is and why it is useful. You will get a chance to run some programs and to examine their parts. You will also be able to look at the data used by the programs.

This chapter also provides a general understanding of how programs work and the things they can do for you. The rest of the book will focus on the APL building blocks you can use to create your own programs.

Before you begin reading this chapter, read the Introduction to this book and the instruction sheets provided along with this system, to learn about getting started in APL.

### PROGRAMS AND DATA

The purpose of learning APL, or any other programming language, is to learn how to write programs. What exactly *is* a program, you might well ask, and what is it good for?

A program is simply a list of instructions telling the computer what to do. It is similar to a recipe for baking a cake—the author of the cookbook gives you a list of things to do to complete the task (bake the cake), and you follow them through in the order given. If you miss a step or mistakenly get the steps out of order, you'll still wind up with something, but it won't be the cake you planned on.

The ingredients you use to bake the cake (flour, eggs, and so on) correspond to the data that a program uses. For a program to do work, it must have something to work *on*, and we call this data. Many programs use numbers as data (as desk calculators do), and others use words and punctuation (as word processors do).

Programs are useful because they can be stored inside a computer, where they can be used over and over again. This means that, once you have a program for figuring your taxes or for playing a game, you can use it over and over and it will always behave the same way. We might get drowsy and make a mistake in the middle of figuring our taxes, but the computer never gets sleepy and won't make those kinds of mistakes.

Once you are familiar with writing programs, you can get the computer to do virtually anything you want with numbers and words. The purpose of this book is to teach you enough APL to write programs to do some interesting things.

## Loading a Workspace

In APL, programs and data are stored in places called “workspaces.” To get a workspace, you must “load” it. This simply activates the workspace so you can work with the programs and data stored in it.

To show you what programs can do, we’ve provided a ready-made workspace called *DEMOAPL*. To get this workspace, enter the command shown below. (By “enter,” we mean type the letters, including the spaces, and then press the RETURN key, which is labeled ENTER on some keyboards.)

```
      )LOAD DEMOAPL
SAVED. . .
```

The system will print a message that includes the word *SAVED* and a date and time. The system then automatically displays the contents of a variable named *DESCRIBE* that describes the other programs in the workspace. (Since the text will scroll up faster than you can read it, press CTRL-S to stop the display, then press CTRL-Q to continue it.)

**Remember:** We’re assuming that you’re using the APL\*PLUS System for the PC on an IBM personal computer and that the *DEMOAPL* workspace (on Disk 4 of the system) is in your default disk drive. (If you don’t know what that means, read the documentation that came with your system.) If you are using a different APL system on a different computer, you may have to enter a slightly different version of the command. The following table shows some examples.

| Command                     | System                    |
|-----------------------------|---------------------------|
| )LOAD MYDISK:DEMOAPL        | APL*PLUS Mac System       |
| )LOAD 1 DEMOAPL             | APL*PLUS PC, Disk Drive B |
| )LOAD /USR/APL/REL3/DEMOAPL | APL*PLUS UNX System       |
| )LOAD 101 DEMOAPL           | APL*PLUS Mainframe System |

If you made a typing mistake when entering the *LOAD* command, you probably got an error message like *WS NOT FOUND* or *INCORRECT COMMAND*.

If this happens, don't worry—mistakes can *never* hurt the computer. After a while, you'll find that error messages are helpful, since they tell you what went wrong. For now though, simply try again by retyping the command. When the *SAVED* message prints, you'll know you're in the right place.

### Running a Program

Once the *SAVED* message prints, you're ready to run some programs. Enter:

```
CALENDAR 9 1982
```

```
SEPTEMBER 1982
```

| SUN | MON | TUES | WED | THUR | FRI | SAT |
|-----|-----|------|-----|------|-----|-----|
|     |     |      | 1   | 2    | 3   | 4   |
| 5   | 6   | 7    | 8   | 9    | 10  | 11  |
| 12  | 13  | 14   | 15  | 16   | 17  | 18  |
| 19  | 20  | 21   | 22  | 23   | 24  | 25  |
| 26  | 27  | 28   | 29  | 30   |     |     |

The program will print a calendar for September (month number 9) of 1982. Try the program again with some other months, such as the month you were born.

Again, if you make a typing mistake, it's no big deal. You'll get some sort of an error message (like *SYNTAX ERROR*) but then you can just try again by entering *CALENDAR* followed by a month number and year.

Now let's do something else. We want to see how the program *works*. That is, you want to look at the parts of the program. So let's list the contents of the program.

```
LIST 'CALENDAR'
```

If nothing is displayed, you probably made a typing error, so just try again. Also, be sure to include the two single quote marks. Leaving a quote mark out may cause your terminal to "freeze up," so just enter another quote if this happens.

The command `LIST 'CALENDAR'` produces the following display.

```

▽ Q←CALENDAR M;ΠIO;T
[1]  ACALENDAR mm yy
[2]  Aprints a calendar of a specified month mm for year YY
[3]  ΠIO←1 ⋄ T←M←M[↑M],1 ⋄ M←1⋄3↑M
[4]  ΠERROR(21033≥ 0 12 1M[3 1])/'NOT BEFORE OCT 1752'
[5]  Q←1001(M[1 3]← 1 0 +⋄ 0 12 τ-3+ 0 12 1M[3 1])[2]
[6]  Q←71M[2]+Q+(1-0.2+2.6×M[1])+(1Q+4)-[1.75×1M[3]+100
[7]  M←(T[2]← 4 6 9 11)+(T[2]=2)×3--/0= 4 100 400 4000 1pT
[8]  Q←'BI6' ΠFMT 6 7 p42↑(Qp0),131-M
[9]  Q←'    SUN    MON    TUES    WED    THUR    FRI    SAT',[1]Q
[10] M←'JANUARY  FEBRUARY  MARCH    APRIL    MAY      '
[11] M←M,'JUNE    JULY      AUGUST   SEPTEMBER OCTOBER  '
[12] M←(12 9 pM,'NOVEMBER DECEMBER ')[T[2];]
[13] M←(M1' ')↑M ⋄ M←M,τT[1] ⋄ M←((2+1(42-pM)+2)p' '),M
[14] Q←' ',[1](35↑M),[1]Q
▽

```

First, notice how *short* the program is. You may have seen programs written in other languages, and they were probably three or four (or even more!) times longer than this. APL is a very *concise* programming language—it allows us to do a lot in a very little space.

Second, notice the numbers on the left-hand side of the program. These numbers tell the computer the order for doing the steps of the program. Just like it was important for us to follow the order of the steps in baking a cake, it's important for the computer to follow the steps of a program in the correct order.

Third, notice the “funny symbols” in the lines of the program. These are APL symbols, and each does a particular job. The rest of this book will focus on explaining what those symbols do and how they can be put together to make a program like *CALENDAR*.

To learn about some other programs in the workspace *DEMOAPL*, enter:

### *DESCRIBE*

You'll see some text describing the different programs in the workspace and how to use them. Since the text is long, we won't show it here. We urge you to try as many of the programs as you like, though, to get a feel for the types of things

---

programs can do. (If you get a *VALUE ERROR* message when you enter *DESCRIBE*, you probably made a typing error. Just try again by typing *DESCRIBE*.)

## LISTING THE NAMES OF PROGRAMS

Now that you've run programs to your heart's content, let's try something new. Let's list the names of *all* the programs in the workspace:

***)FNS***

The command is *)FNS* because programs are often called *functions* in APL.

We're not going to show all of the list printed by *)FNS*, since it may be different on different systems. The list may run for several lines though.

If you like, you can list these programs (functions) like you did *CALENDAR*:

***LIST 'name'***

Don't enter *LIST NAME* —rather, enter *LIST* and then one of the function names included in the list printed by *)FNS*. Be sure to put quotes around the name.

## LISTING THE NAMES OF VARIABLES

We can also look at the data in the workspace. Enter the command

***)VARS***

(Again, we won't print the entire list.) What prints is a list of names again, but this time it's a list of *data* names, separated by blanks. Data is stored in *variables*—called that because the data inside them can vary. The programs in workspace *DEMOAPL* use the data stored in these variables to produce calendars and do all the other things mentioned in the *DESCRIBE* listing.

To see what's inside a variable, you just enter its name:

***MON***

9

You can look at the contents of any of the variables in the list printed by *)VARS* by just entering its name this way. Be sure to enter only *one* name at a time.

Probably the contents of most of these variables will be pretty mysterious to you—a few odd numbers and words. It's not important now that you know what the numbers and words are used for though. We only want you to realize that this data is stored in the computer for programs to work with. In the next chapter, you'll learn how to store your own data and work with it in different ways.

## **SUMMARY**

1. A program is a list of instructions that tells the computer how to do some work. A program is also called a “function” in APL.
2. A program does its work using data.
3. Data is stored in something called a “variable.” You can see what's stored in a variable by entering the variable's name.
4. A “workspace” is a place where programs and data are stored.
5. You can list the names of the functions and variables in a workspace with the commands `)FNS` and `)VARS`.



## Chapter 2

# Simple Arithmetic, Error Correction, and Data Storage

---

In this chapter, you will learn how to do arithmetic with APL using single numbers, lists of numbers, and tables of numbers. You'll also learn how to store these numbers in the computer's memory. You'll be introduced to the shape, reshape, and catenate functions, the first two of which are unique to APL.

At the end of this chapter, as at the end of most of the following chapters, there are sets of exercises for you to work on. We urge you to work at least some of the problems, because that's one way to really gain experience with APL. Solutions and explanations are provided in Appendix A.

### USING APL LIKE A DESK CALCULATOR

Now that we've seen some things we can do with APL, we need to back up a little and explore the tools APL provides us with to do those things.

First, enter the command:

```
      )CLEAR  
CLEAR WS
```

#### clearing the workspace

The system responds with *CLEAR WS*, which means you have an empty, unused workspace. If you enter:

```
      )FNS
```

or

```
      )VARS
```

you'll see that there's nothing there.

#### immediate execution mode

In its usual state, the APL system can be used like a desk calculator. Try entering:

```
      128+307  
435  
      832-299  
533
```

```
      6 6 ÷ 2
3 3
      4 × 1 2
4 8
```

We call this “immediate execution mode,” since APL immediately gives us an answer to what we type in.

The questions we just asked APL were pretty easy. Let’s try something a bit harder:

```
      2 3 ÷ 7
3 . 2 8 5 7 1 4 2 8 6
      3 4 6 7 4 ÷ 7
4 9 5 3 . 4 2 8 5 7 1
```

## HANDLING LISTS OF NUMBERS

What about doing something with a list of numbers? Try:

```
      7 5 3 ÷ 7 4 2
1 4 9 5
```

(Be careful to leave at least one space between each of the numbers.) What happened? The first number on the left of the plus sign was added to the first number on the right, the second number on the left to the second number on the right, and so on.

## vectors and scalars

One of the nicest features of APL is that a single function can work on long lists of numbers. A list of numbers, like 7 5 3 in the example above, is called a “vector,” which is just shorthand for “a bunch of numbers.” If you want to refer to a single number, you can call it a “scalar,” which is a little shorter than saying “a single number.”

Let’s try some more:

```
      7 5 3 × 7 4 2
4 9 2 0 6
      7 5 3 ÷ 7 4 2
1 1 . 2 5 1 . 5
      8 6 - 6 4
2 2
```

But then try:

```

      4 5+1 2 3
LENGTH ERROR
      4 5+1 2 3
          ^
  
```

We got an error message, because two vectors must be of equal *length* (that is, contain the same number of elements) in order to be combined. Since the statement cannot be executed, APL tells you why.

### operations with vectors and scalars

What about mixing vectors with scalars? Try:

```

      8 10 12+3
11 13 15
  
```

We can see that the single number (scalar) is added individually to each number in the vector. So if a scalar appears on one side of a function symbol (like + or ÷) and a vector is on the other side, the scalar is added to (or subtracted from, multiplied by, and so on) each number in the vector. Try this with the other arithmetic functions we've learned and you'll see that the results are similar.

### NEGATIVE NUMBERS

In regular arithmetic notation, the same symbol is used to mean the operation subtraction (2 - 4) and the sign for a negative number (-2).

In APL, the subtraction function is done with the familiar "middle minus" (hyphen) sign:

```

      10-5 4 3
5 6 7
  
```

### high minus

while a negative sign is the "high minus." The high minus is usually located on the 2 key; type it by pressing SHIFT, ALT, or OPTION with 2.

```

      -3+8
5
  
```

If you put the hyphen to the left of a number, it turns the number into a negative number:

- 4  
-4

Or, it turns a negative number into a positive one:

- - 3  
3

But, the entry:

- - 4  
SYNTAX ERROR  
- - 4  
^

gives a *SYNTAX ERROR* because you can't have a high minus sign separated from a number it's part of. It has to be right next to the number.

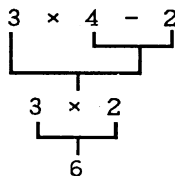
## ORDER OF EXECUTION

Let's try two arithmetic functions in the same line. Enter:

3 × 4 - 2  
6

## Right-to-Left Execution

Did you expect the answer to be 10? That's because in regular arithmetic, multiplication is performed before addition and subtraction. APL has so many different functions that it would be confusing to have a set of rules about which are used before each other. APL uses one easy rule—it always analyzes a statement right to left. That is, it evaluates the function at the far right of the line *first*, then moves left to the next function. APL evaluated the example above like so:



Let's try a few more examples:

```

      6 ÷ 2 + 1
2
      8 + 3 × 2
14
  
```

APL also works this way for statements with a large number of functions in them. If we want to find the sales tax for a purchase with three different items (assuming a tax rate of 5%), we can enter:

```

      .05 × 1.32 + 4.10 + .78
0.31
  
```

### Use of Parentheses

The order of execution can be changed by using parentheses to surround the expression we want executed first. The contents of the parentheses are treated as a single value, so the system must calculate this value before using it.

```

      ( 6 ÷ 2 ) + 1
4
      6 ÷ 2 + 1
2
      ( 8 + 3 ) × 2
22
      8 + 3 × 2
14
  
```

If we want to find the number of square feet in three rooms, we can multiply the length and the width of each room, and then add them up:

```

      ( 12 × 15 ) + ( 20 × 15 ) + ( 15 × 12 )
660
  
```

### ERROR CORRECTION

So far, so good. We trust that you have been able to get through most of the examples up to this point without any major problems. You've probably made some typing errors and received some error reports, though (like *LENGTH ERROR*), so now you're ready to learn about correcting errors (also called "input line editing").

**Interrupting  
Input**

If you type a line incorrectly and notice it *before* pressing RETURN (or ENTER), you can fix the mistake. The simplest thing to do is to “interrupt” your input. This causes the computer to ignore the entire line and return you to the six-space indent. To interrupt input, you press a key or key combination that sends a special signal to the computer. On the APL\*PLUS System for the PC, you press CTRL-BREAK. Other APL systems use slightly different techniques; for example, the APL\*PLUS System for the UNIX environment has an UNDO key. If you don’t know which key to use, refer to your system documentation or consult an experienced user.

**Untype**

If you want to delete characters at the end of a line, you can use another technique. Hold down the BACKSPACE (UNTYPE) key until you have erased all the characters you want. Then you can type in the correct information, keeping the earlier part of the line.

**Delete**

Another way to delete characters is by using the DELETE key, which is found on most personal computers. Use the cursor movement keys (usually located on the numeric pad) to move to the mistake, then press the DELETE key. The character *above* the cursor will be the one that is deleted.

Take a few minutes to practice these techniques by typing in some long lines and making changes in them. You will find that these tricks come in handy later on.

**STORING  
DATA IN  
VARIABLES**

By now, you’ve learned that doing arithmetic in APL is similar to doing it on a desk calculator. Most calculators allow you to store data in one or more “memories.” We saw in Chapter 1 that APL stores data in storage places (memories) called variables. We were able to read the data in the variables.

In APL, you store data in a variable by “assigning” the data to a name. The left-pointing arrow (“assignment arrow”) does this:

```
NUMBER←1
SALES←15.80 17.40 100.20
GRADES←78 97 81 85 92
```

As we've seen, we can display the data stored in variables by entering their names:

```
      NUMBER
1
      SALES
15.80 17.40 100.20
      GRADES
78 97 81 85 92
```

Variable names can be made of letters and numbers. The name always begins with a letter.

#### using variables with functions

You can use variables just as you would use the numbers they contain. For example, suppose we wanted to find out how much the tax on three items sold (contained in *SALES*) would be, with a 5% tax rate:

```
      SALES * .05
0.79 0.87 5.01
```

If we wanted to double our sales next month:

```
      2 * SALES
31.6 34.8 200.4
```

Variables can be added to other variables as well (or subtracted, divided, and so on).

```
      NUMBER + SALES
16.8 18.4 101.2
```

Variables retain their values until we assign new data to them.

```
      NUMBER
1
      NUMBER ← 2
      NUMBER + NUMBER
4
      NUMBER
2
      NUMBER ← 1
      NUMBER
1
```

We can use a variable and reassign it in the same line:

```
NUMBER←NUMBER+4
NUMBER
```

5

## SHAPE FUNCTION

APL provides a useful function to help us figure out how many numbers are in a variable. Use the symbol  $\rho$  (rho) with a variable name after it:

```
5           $\rho$ GRADES
          GRADES
78 97 81 85 92
```

This function is called “shape,” since it gives the shape of the variable (how many elements it contains). This may seem so easy that it is not needed. But you can store many numbers in an APL variable. In fact, you can keep adding numbers until the line is filled and the computer skips to the next line:

```
          SALES1←200 175 125.50 421.75 89 23
304 89.95 256.72 78 21 78.23 75.21
           $\rho$ SALES1
```

13

(In the example above, the computer may not have to jump to the next line for you if you’re using a wide terminal, but you get the point.)

Thousands of numbers can be stored in a variable.

## CATENATE FUNCTION

Another APL function useful for dealing with variables is “catenate.” This function joins variables together to make one large variable. Comma (,) is the symbol used for catenation. Enter:

```
          SALES2←SALES, SALES1
           $\rho$ SALES
3
           $\rho$ SALES1
13
           $\rho$ SALES2
16
```



```
SALES2
15.8 17.4 100.2 200 175 125.5 421.75 89 23 304
      89.95 256.72 78 21 78.23 75.21
```

```
ONE←1
TWO←ONE, ONE
TWO
1 1
  ρTWO
2
```

You can see that shape ( $\rho$ ) is very useful for counting the elements in a long vector produced using catenate.

## WORKING WITH TABLES OF NUMBERS

We've seen that both single numbers (scalars) and groups of numbers (vectors) can be used with arithmetic functions and that they can be assigned to variables. Another useful form for data is a table, such as the sales from different stores in different months. (*Don't* enter the example below—it's just for illustration.)

|                | Jan   | Feb   | Mar   |
|----------------|-------|-------|-------|
| Elm St. Store  | 10000 | 11000 | 10525 |
| Lee Ave. Store | 9000  | 9250  | 10000 |
| Pine Ln. Store | 6000  | 6100  | 6125  |
| Ash St. Store  | 8250  | 10500 | 7500  |

## Reshape Function

A "matrix" is a table of data like the numbers in this chart. It has two dimensions: rows (going across) and columns (going down). We can make a table in APL using our old friend rho ( $\rho$ ) in a new way. Enter:

```
STORESALES←4 3ρ10000 11000 10525
9000 9250 10000 6000 6100 6125 8250 10500
7500
```

(Again, the computer may or may not skip to a new line for you as shown above, depending on the width of your terminal or screen.)

```

      STORESALES
10000 11000 10525
 9000  9250 10000
 6000  6100  6125
 8250 10500  7500

```

The 4 3 on the left of  $\rho$  tells APL that we want a table (matrix) with 4 rows and 3 columns. We call  $\rho$  “reshape” when it’s used with numbers on both the left and right sides. It reshapes the data on its right into the shape given on its left. We can use reshape to make vectors:

```

      Q←4 ρ 7
      Q
7 7 7 7

```

(This expression creates a vector with four elements.) We can also create matrices with only *one* column:

```

      R←3 1 ρ 5 10 15
      R
5
10
15

```

or with only one row:

```

      Z←1 4 ρ 7
      Z
7 7 7 7

```

Z looks exactly the same as a vector, but it has different shape:

```

      ρ Z
1 4
      ρ Q
4

```

As you can tell from the example above, the shape function (rho with data on the right side only) also works with matrices:

```

      ρ STORESALES
4 3

```

---

Asking for the shape of a matrix tells us how many rows (4) and how many columns (3) the matrix has.

### Arithmetic with Tables

Let's use our new-found skill with matrices to do a little more arithmetic. Let's say we have a matrix of salaries of salesmen at three stores:

|         | Emp 1 | Emp 2 | Emp 3 |
|---------|-------|-------|-------|
| Store 1 | 18000 | 17500 | 16500 |
| Store 2 | 19000 | 21000 | 20000 |
| Store 3 | 20000 | 17500 | 19500 |

In APL, we enter:

```
SAL←3 3p18000 17500 16500 19000
21000 20000 20000 17500 19500
```

How much will they get if they receive a 15% bonus? Enter:

```
BONUS←SAL×.15
BONUS
2700 2625 2475
2850 3150 3000
3000 2655 2925
```

What if we need to subtract (heaven forbid!) \$500 from each salary?

```
SAL-500
17500 17000 16000
18500 20500 19500
19500 17000 19000
```

### matrices and scalars

We can see that our arithmetic functions use matrices and scalars together. This means the functions are even more powerful than we thought. Again, the single number is used with each element of the matrix.

**matrices  
and vectors**

However, using the arithmetic function you've learned so far, matrices and vectors *cannot* be combined:

```

      SAL+8 9 7
RANK ERROR
      SAL+8 9 7
      ^

```

The rank (or number of dimensions as shown in the shape) is not the same. The problem is that such statements are ambiguous. Do we want to add 8 to all the numbers in the first *row* or to all the numbers in the first *column*? Since the statement can't be executed, APL tells you why.

**matrices  
and matrices**

Matrices *can* be used with other matrices, however, if they are the same size (shape). If we want to find the total salaries for the year for our employees, including the 15% bonus:

```

      SAL+BONUS
20700 20125 18975
21850 24150 23000
23000 20125 22425

```

**FUNCTION  
SYMBOLS  
WITH DOUBLE  
USES**

You're probably wondering why we were able to use the rho symbol ( $\rho$ ) to do two different things. At the beginning of the chapter, we used it by itself with a variable on its right:

```

      ρSALES2
16

```

Later, we used it with numbers on its right and left:

```

      4 1 ρ1 2 3 4
1
2
3
4
      3 ρ2
2 2 2

```

**Arguments**

In APL terminology, the things (data) that appear to the left and right of a function are called “arguments.” We have learned that  $\rho$  can be used with one argument (in particular, a *right* argument) or with two arguments (left and right arguments). Many APL symbols may be used this way. Another way of expressing this is that they can be either “monadic” (having one argument) or “dyadic” (having two arguments). They have different results in each case.

**Monadic and Dyadic Functions**

The following table is a summary of the rules for combining scalars, vectors, and matrices using the arithmetic functions we’ve learned so far. In the table, “fn” stands for any of the arithmetic functions (+, −, ÷, ×) and the arrow (←) identifies the result of the operation. For example, “scalar ← scalar fn scalar” means that when you use any of these functions with two scalars, the result will be a scalar too.

**Rules for Combining Scalars, Vectors, and Matrices**

| <b>Result</b> |   | <b>Left Argument</b> |    | <b>Right Argument</b> |
|---------------|---|----------------------|----|-----------------------|
| scalar        | ← | scalar               | fn | scalar                |
| vector        | ← | scalar               | fn | vector                |
| vector        | ← | vector               | fn | scalar                |
| vector        | ← | vector               | fn | vector                |
| matrix        | ← | scalar               | fn | matrix                |
| matrix        | ← | matrix               | fn | scalar                |
| matrix        | ← | matrix               | fn | matrix                |

Many of the functions we will learn in the following chapters also obey these rules.

**note**

Before you go on to the next section, you may want to save (store) the variables you’ve created so far so you can use them later. To do this, enter:

**⋄SAVE MYWORK**

If you are using a personal computer and you get a *DISK FULL* or similar error message when you use this command, try inserting a blank, formatted disk in the disk drive. Then enter the command again.

When you sign on next time, you can retrieve your work by entering:

**`)LOAD MYWORK`**

If you are using a personal computer, be sure that the correct disk is in the disk drive.

Remember that you can list the variables by entering:

**`)VARS`**

## **SUMMARY**

1. APL uses the familiar arithmetic functions very much like we would use them by hand or with a calculator.
2. One exception is that APL uses a consistent order of executing functions on a line: always right to left. You can change the order by using parentheses to enclose those operations you want done first.
3. APL uses its own symbols to perform other operations. We learned about three symbols: the left arrow ( $\leftarrow$ ), rho ( $\rho$ ), and comma ( $,$ ).
  - a. The left-pointing arrow stores numbers by “assigning” them to a name. Scalars, vectors, and matrices can all be assigned to variables.
  - b. The rho symbol ( $\rho$ ) can be used two ways:
    - to measure the size of something (shape)
    - to create a matrix or vector (reshape)
  - c. The comma is used to join (catenate) variables together.
4. Many APL symbols serve two purposes. These symbols can be used with a right argument only (that is, data on the right side only) or with both left and right arguments (data on both sides of the function symbol).

Now it's time for some practice problems. Try to figure out in advance what the answer to each problem is. Then enter the problem into the computer and see what happens. Answers are given in Appendix A. If you miss a lot of the answers at first, don't worry. Things will become much clearer as you continue to practice.

Some of the exercises will result in error messages rather than actual answers. Explanations of these messages appear in Appendix A along with the solutions.

## EXERCISES

- ✓A. 1.  $3+1$  2 3                      7.  $3^{-2}$                       13.  $5(2+6)$   
       2.  $6\div2$                       8.  $7\ 4\ 5-2$                       14.  $4\div1\ 2$   
       3.  $2-4$                       9.  $10\div0$                       15.  $-^{-}8$   
       4.  $3\times2\ 4$                       10.  $5\times2+6$                       16.  $4\div2\times1+1$   
       5.  $3\ 3\times2\ 4$                       11.  $4\ 6+3\ 5\ 7$   
       6.  $4+$                       12.  $(5\times2)+6$

✓B. Create three variables as shown below:

```
A←2
B←1 4 7
C←2 3ρ0 1 2 3 4 5
```

Use these variables to do the exercises in this section.

- |          |                  |                 |
|----------|------------------|-----------------|
| 1. $A+A$ | 5. $C+C$         | 9. $2\ 4-B$     |
| 2. $A+B$ | 6. $B+C$         | 10. $2\times B$ |
| 3. $A+C$ | 7. $4+C$         |                 |
| 4. $B+B$ | 8. $9\ 10\ 11-B$ |                 |

✓C. Use the following variables for the problems in this section:

```
V1←0 0 1
V2←1 2 3 4
V3←2 1
```

- |                  |                  |                |
|------------------|------------------|----------------|
| 1. $\rho V1$     | 5. $\rho V1, 5$  | 9. $V3\rho V1$ |
| 2. $\rho V2$     | 6. $V2+V1, 5$    | 10. $5, V1$    |
| 3. $\rho V1, V2$ | 7. $3\rho 4$     |                |
| 4. $V1, 5$       | 8. $3\ 3\rho V3$ |                |

D. 1. You are at the grocery store and you buy the following items:

- 12 potatoes at 20 cents apiece
- 6 pears at 30 cents apiece
- 2 avocados at 50 cents apiece

Write an APL expression using two vectors to calculate the total cost of each item.

2. Write an APL expression to find the yearly interest on four savings accounts, given annual interest rates of 5%, 5.25%, 5.5%, and 6%. The four accounts contain \$600, \$700, \$1000, and \$950, respectively. (HINT: .05 is the same as 5%.)

√3. To find the grades for students who scored 48, 39, 41, and 35 out of a possible 55 points, use the following expression:

$$100 \times 48 \ 39 \ 41 \ 35 \div 55$$

The division part of the expression turns the grades into a ratio, and multiplying the ratios by 100 turns them into percentages.

Write an APL expression to find the percentage grades for students who scored 142, 167, 161, and 128 points out of a possible 175 points.

4. Write an APL expression to find the tax due on a taxable income of \$7216. Assume the tax is \$2116 plus 25% of the amount of taxable income which exceeds \$6000.

√5. Your store sells hammers, wood, and nails. Last week, you sold 12 hammers, 9 pieces of wood and 3 nails; this week you sold 11, 12, and 6 items. Store this data in one matrix with one row for each week. The prices for the three items were 3.25, 1.79, and 2.55 last week, but this week you raised them to 3.45, 1.85, and 2.75. Store this data in a second matrix. Now write an APL expression which calculates the total sales revenue by item for both weeks.



6. The vector *GRADES* contains students' grades from last semester (90, 82, 79, 92). Put the grades for this semester's students (93, 87, 80, 90, 78, 85) into the variable *GRADES* without losing last semester's grades. (*HINT*: Introduce a new variable *OLDGRADES*.)
7. You are the manager of a small store and you take inventory at the end of the month. You store the number of items in inventory in a variable named *QUANT*:

*QUANT*←210 51 34 27

You store the cost of a single unit of each item in the variable *COST* :

*COST*←2.25 15.17 20.72 5.75

Write an APL expression that calculates the cost of inventory and assigns it to a variable named *INVCOST*.

## Chapter 3

# Character Data, More Arithmetic, and More APL Functions

---

In this chapter, you will learn:

- how words and sentences are handled and stored using APL
- how to use the APL functions “maximum” and “minimum”
- how to use the APL functions “plus reduction” and “plus scan” to total numbers and produce a running total
- how to pick out single elements of stored data using the APL function “index”
- how to produce strings of numbers using the APL functions “count” and “roll”

Learning about these functions will help you to do some useful things with bank balances, such as getting APL to:

- find the ending balance
- keep a running total of deposits and withdrawals
- find the largest and smallest deposits and withdrawals

### CHARACTER DATA

So far, we’ve only dealt with numbers. Numbers are known as “numeric data” in computer jargon. One of the special powers of APL is that it can work with words and sentences as well. In this capacity, APL is similar to a word processor rather than to a desk calculator. Letters, symbols, words, and sentences are known as “character data” in APL.

### using single quotes

In APL, character data is distinguished from numeric data by enclosing the characters in ‘single quotes.’ The single quote is usually found on the K key on an APL keyboard, as a combination with SHIFT, OPTION, or ALT. Enter:

```
      'HELLO'  
HELLO
```

Anything you put inside single quotes will be displayed by the system exactly as you entered it.

Thus, we can learn the shape of a group of characters:

```

5      ρ 'HELLO '

```

Blanks are also considered elements.

```

8      ρ 'HI  THERE '
6      ρ 'I   SPY '

```

(In the second example, we purposely left *two* blanks in between the two words.)

Special APL characters can also be used as character data. On most APL systems, you press the SHIFT key to produce some:

```

      '***  ΔΔΔ '
***  ΔΔΔ
      ' "WHAT WAS THAT? " '
"WHAT WAS THAT? "

```

#### “open quote” error

If you type only one quote, APL considers it an “open quote” error. Different APL systems react differently to this error; however, most catch the error and refuse to go on until you correct it. To correct the error, you can usually:

- interrupt the system; for example, press CTRL-BREAK, UNDO, or COMMAND-PERIOD
- type a second quote and press ENTER

#### using quotes in character data

If you want to include a single quote as a character in a sentence, double it.

```

      'WHAT' 'S UP? '
WHAT'S UP?
      'DON'T SAY AIN'T.'
DON'T SAY AIN'T.

```

APL translates the two quote marks back to a single one. The general rule is that you must have an *even* number of quotes in a line to avoid the “open quote” problem.

**character data  
that looks like  
numbers**

Digits can also be used as character data.

```
      ' 7 3 '
7 3
      ' 1 + 1 '
1 + 1
```

If you use digits as character data though, they cannot be used to do arithmetic. APL does not treat them as numbers.

```
      3 + ' 8 '
DOMAIN ERROR
      3 + ' 8 '
      ^
```

You get a *DOMAIN ERROR* report, because it's not in the “domain” of the plus function to add numbers to characters. Remember, only numbers and numbers can be added together. Entering `3 + ' 8 '` is just like entering `3 + ' * '`—it's a meaningless statement. You can enter something like:

```
      ' 4 × 1 2 '
4 × 1 2
```

because everything between the quotes is considered character data, even the times sign.

**character vectors**

Groups of characters in a line, such as the ones we've been entering, are called “character vectors.”

**using variables  
with character  
data**

Character vectors can be assigned to variables just like numeric vectors.

```
      MESSAGE ← ' HELLO , WORLD . '
      MESSAGE
HELLO , WORLD .
```

**using catenate  
with character  
data**

Character vectors can be joined together with other character vectors using the catenate function:

```
PART1←'HELLO.  '
PART2←'WHAT' 'S YOUR NAME?'
PART1,PART2
HELLO.  WHAT'S YOUR NAME?
```

**Format  
Function**

In most systems, character vectors can't be connected directly to numbers:

```
'THE TOTAL IS ',27
DOMAIN ERROR
'THE TOTAL IS ',27
^
```

Just as it wasn't in the domain of the plus function to add numbers and characters, it isn't in the domain of the catenate function to combine numbers and characters. But we can join the data after converting the number to character data using a function called "format." The symbol for format is  $\text{⍒}$ . (It's usually found on the single/double quote key; you generally type it as a combination with SHIFT, ALT, or OPTION.) The effect of the format function is to turn numbers into characters; they can then be catenated to other characters. Let's try:

```
'THE TOTAL IS ',⍒27
THE TOTAL IS 27
```

**Character  
Matrices**

Character data can be reshaped by our old friend  $\rho$  to form character matrices.

```
TEXT←3 3⍴'HOWAREYOU'
TEXT
HOW
ARE
YOU

ANSWER←2 6⍴'FINE, THANKS'
ANSWER
FINE,
THANKS
```

Notice that you have to count out the characters and insert blanks in the proper places so that shorter lines will have blanks at the end. If you don't, the shorter lines will be filled with data you meant to appear on the next line.

```
      NAMES←3 4ρ'JILLJIMBOB'  
      NAMES  
JILL  
JIMB  
OBJI
```

You've told APL to create a "box" of characters, three characters high and four characters across, but you've only given it ten characters to work with. So it uses characters in groups of four (such as *JILL* and *JIMB*). When it reaches the end of the string, it goes back to the beginning to complete your request. The *OBJI* in row 3 is composed of the *OB* from the end of the string and the *JI* repeated from the beginning of the string.

We can fix our names list by inserting some blanks:

```
      NAMES←3 4ρ'JILLJIM BOB '  
      NAMES  
JILL  
JIM  
BOB
```

We'll do more with character data in the Indexing section of this chapter and in Chapter 4.

## MAXIMUM

The "maximum" function chooses the larger of two numbers. In particular, it compares the number on its right side to the number on its left side to see which is larger. The APL symbol for maximum is  $\Uparrow$ , usually found on the S key.

Enter:

```
      8 $\Uparrow$ 10  
10  
2.1 $\Uparrow$ 2  
2.1
```

Like all APL functions, it works with data stored in variables:

```

ONE←1
TWO←2
ONE↑TWO
2

```

APL looks at the values stored in the variables to see which is larger; then it displays the larger value.

As you probably guessed, the result of maximum can be stored in a variable. (This is true of *all* built-in APL language functions.)

```

B←7↑19
B
19

```

#### maximum with vectors and matrices

The examples below clearly show that maximum obeys the rules laid out in the chart at the end of Chapter 2.

```

1↑3 1 0 6
3 1 1 6
5 6 3 1↑6 8 2 .5
6 8 3 1
TABLE←2 3ρ1 2 3 4 5 6
TABLE
1 2 3
4 5 6
3↑TABLE
3 3 3
4 5 6

```

Let's do something practical with this function. Suppose you have a number of orders for warehouse items:

```

ORDERS←2.71 3.25 10.17 7.32 3.05 1.79

```

You've set the minimum purchase at \$3.00, however. So you write an APL statement that will give you the greater of \$3.00 or the amount of the order.

```
BILL←3∇ORDERS
BILL
3 3.25 10.17 7.32 3.05 3
```

## MINIMUM

The minimum function compares two numbers and takes the smaller of the two. The symbol for minimum is  $\mathbb{L}$ , usually found on the D key. The shapes of the symbols make it easy to remember which is which ( $\mathbb{L}$  means larger,  $\mathbb{L}$  means smaller). Enter:

```
8      8∇10
8
1 3 2  1 3 4∇4 3 2
3∇TABLE
1 2 3
3 3 3
```

You could use this function if you had a number of orders for units of a discounted item, but you had set a limit of 10 items per customer.

```
DISCORDERS←12 15 7 9 17
DELIVER←10∇DISCORDERS
DELIVER
10 10 7 9 10
```

## REDUCTION

What if we wanted to find the largest number in a string of numbers? We could enter:

```
10∇12∇2∇72∇16
72
```

## Maximum Reduction

This is rather time-consuming though. So APL provides a quick and easy way to do the same thing:

```
∇/10 12 2 72 16
72
```



All we did was combine the maximum symbol with a slash, and this had the same effect as inserting the symbol in between each pair of numbers. We call this "reduction," and since it's used here with the maximum function, its full name is "maximum reduction." Reduction can be used with variables:

```

      ⍒ / ORDERS
10.17
      ⍒ / DISCORDERS
17
    
```

### Plus Reduction

Reduction works with other functions as well. Enter:

```

      + / 2 3 4
9
    
```

What happened? Again, it's the same as entering

```

      2 + 3 + 4
9
    
```

### Times Reduction

We can also use reduction with multiplication:

```

      × / 2 3 4
24
    
```

### Minimum Reduction

And, of course, with minimum:

```

      ⍒ / DISCORDERS
7
    
```

Actually, you can use reduction with any of the arithmetic functions we've learned so far. For right now though, the most important ones for you to know about are plus reduction, maximum reduction, and minimum reduction.

### Reduction with Matrices

What about using reduction on a matrix? Let's try it on our variable *TABLE*.

```

      TABLE
      1 2 3
      4 5 6
      + / TABLE
6 15
    
```

\*  $A \leftarrow 1 \ 3 \ 7 \ 4$

$\leftarrow + / A$        $\leftarrow - / A$

$\oplus$        $\ominus$        $= 1 + 3 + 7 + 4$

What happened? It *did* work; APL didn't reject it. What APL did was add the numbers along the *rows*:

$$\begin{array}{rcccccl} 1 & + & 2 & + & 3 & = & 6 \\ 4 & + & 5 & + & 6 & = & 15 \end{array}$$

**reduction  
across the  
columns**

We can make APL add up the numbers along the *columns* by entering a statement like this:

```
5 7 9      + / [1] TABLE
```

This works because in our first example, APL added up the numbers along the *last* dimension (the columns), producing one answer for each row. Now, we're asking APL to add up the numbers along the *first* dimension (the rows) to produce one answer for each column. This is the meaning of [1] in the statement. You can also write + / [2] TABLE to get APL to add along the rows:

```
        + / [2] TABLE  
6 15
```

But that's really just extra (and unnecessary) typing, since the system assumes you mean the *last* dimension if you don't specify a dimension.

How would we find the grand total of the numbers in a table? The statement

```
        + / TABLE  
6 15
```

gives us the result of adding up the two rows, so we could get the grand total by adding those:

```
        + / 6 15  
21
```

But we can do this all in one statement by writing:

```
+ / + / TABLE
21
```

## SCAN

So now we know how to find a grand total for the numbers in a vector or a matrix. What if we want to find a *running* total, though, such as for a checkbook? APL provides “scan,” something similar to reduction, for this purpose. Its symbol is the “backslash” (\). Enter:

```
ORDERS
2.71 3.25 10.17 7.32 3.05 1.79
+ \ ORDERS
2.71 5.96 16.13 23.45 26.5 28.29
```

The result of this function shows the first number in the vector, then the sum of the first and second numbers, then the sum of the first, second, and third, and so on. The last number in the result (28.29) is the grand total—the same thing we get with `+ / ORDERS`.

```
+ / ORDERS
28.29
```

You probably guessed that you can use scan with other functions, just like reduction:

```
× \ 2 3 4
2 6 24
```

Scan can also be used with maximum, minimum, and other functions, but “plus scan” is the most useful for us to know at this point.

## INDEXING

Let’s forget about arithmetic for a moment and think about our stored data. What if we wanted to find the third element in *DISCORDERS* without having to count? APL provides a handy function for this, called “index.” The symbols used for index are the left and right brackets ([ and ]). The position of the element you’re interested in (which could also be called the “index” of the element) goes in between the two brackets. Enter:

```
12 15 DISCORDERS
      7 9 17
      DISCORDERS[3]
7
```

Remember, variables can hold hundreds or even thousands of elements, so this function can really be useful.

Indexing also works with a vector of numbers:

```
      ORDERS[1 3 5]
2.71 10.17 3.05
```

Indices can be duplicates:

```
      ORDERS[2 2 2]
3.25 3.25 3.25
```

You can also use indexing to select the data in any order.

```
      VECT←9 10 11
      VECT
9 10 11
      VECT[3 2 1]
11 10 9
```

### Using Index with Variables

As we might expect, the result of indexing can be assigned to a variable.

```
      BACKWARDS←VECT[3 2 1]
      BACKWARDS
11 10 9
      VECT
9 10 11
      NUMBER←VECT[2]
      NUMBER
10
```

### Replacing Individual Elements

Notice that you can use indexing together with assignment to replace individual items in a vector:

```
      VECT
9 10 11
```

```

      VECT[2]←3.14
      VECT
9 3.14 11
      VECT[3 1]←22 -44
      VECT
-44 3.14 22

```

## Indexing into Matrices

What about finding a number in a matrix? Remember, we have two dimensions in a matrix. So we must tell APL both the row *and* the column we're looking for.

```

      TABLE
1 2 3
4 5 6
      TABLE[2;1]
4
      TABLE[1;1]
1

```

The first index is the number of the row; the second index is the number of the column. The two indices are separated by a semicolon (;).

You can also use more than one number when you're indexing into a matrix.  
Enter:

```

      TABLE[1 2;1]
1 4

```

Here we looked at rows 1 and 2 and took the number in the first column of each. To get the first two rows *and* the first two columns:

```

      TABLE[1 2;1 2]
1 2
4 5

```

If we leave out one of the indices, this means “select all of the numbers in the specified row or column.”

```
TABLE[1;]
1 2 3
TABLE[;3]
3 6
```

Again, the results of these operations can be assigned to variables.

```
LITTLETABLE←TABLE[1 2;1 2]
LITTLETABLE
1 2
4 5
```

You can also change individual elements of a matrix using indexing and assignment.

```
TABLE[1;1]←25
TABLE
25 2 3
4 5 6
TABLE[2;2 3]←-2 1.5
TABLE
25          2          3
4          -2         1.5
```

What happens when you ask for an element that’s not there? That is, what if you ask for the seventh element of *ORDERS*, but there are only six elements in *ORDERS*?

```
INDEX ORDERS[7]
ERROR
ORDERS[7]
      ^
      ρORDERS
6
```

APL won’t accept the statement, but as usual, it displays a message pointing us to the problem—the index, in this case.

## Indexing with Character Data

You may not have thought of using indexing with character data, but it works with character data just as it does with numeric data.

```

MSG←'HI THERE'
MSG[1 2]
HI
MSG[5 6 7]
HER
NAMES←3 4ρ'BILLJEANDAN '
NAMES
BILL
JEAN
DAN
NAMES[1 2;1]
BJ
NAMES[3;1 2 1 2]
DADA

```

## COUNT

It's often useful to be able to work with a group of numbers without having to type them all in yourself. APL provides a handy function called "count" that generates a string of consecutive whole numbers. Its symbol is  $\iota$  (iota). Let's see how it works:

```

      ⍲7
1 2 3 4 5 6 7
      ⍲12
1 2 3 4 5 6 7 8 9 10 11 12

```

This function starts at one and counts up to the number we say. Count doesn't work with negative numbers or decimal numbers, though.

```

      ⍲-4
DOMAIN ERROR
      ⍲-4
      ^
      ⍲9.7
DOMAIN ERROR
      ⍲9.7
      ^

```

These types of numbers are not in the domain of count, that is, count is not defined for use with them. Almost any sequence of numbers can be created using iota, however.

```
      .5+ι4
1.5 2.5 3.5 4.5
      2×ι3
2 4 6
      -5+ι6
-4 -3 -2 -1 0 1
```

Count can also be very useful when combined with the index function. Suppose we want the first four elements of *ORDERS*. We can enter:

```
      ORDERS[1 2 3 4]
2.71 3.25 10.17 7.32
```

But we can also enter:

```
      ORDERS[ι4]
2.71 3.25 10.17 7.32
```

You can see that this would save a *lot* of typing if we want the first 100 elements of a 1200-element vector!

## ROLL

Sometimes it's useful to generate a list of *random* numbers instead of consecutive ones. The function “roll,” using the question mark symbol (?), generates a random number between 1 and the number you give it.

```
      ?10
4
      ?10
7
      ?6 6
1 5
```

(Since roll generates *random* numbers, you will probably get different answers than those shown above.) The last example simulates the rolling of two dice—hence, the name of the function. Roll can also be combined with other number generators:



```

      ? 1 5
1 2 1 1 3
      ? 3 2 ρ 6
1 3
5 5
4 1

```

~~2 2 2 5~~  
Real - Scalars Out: 3 ? 5  
 1 4 3

In the second example, reshape creates a matrix of 6s (using 3 2 ρ 6), then roll generates six random numbers, each between 1 and 6.

**note**

Remember to store your variables if you want to use them later. If you're using a personal computer, be sure the correct disk is in the disk drive. Enter:

**)SAVE MYWORK**

You can use any name you wish instead of *MYWORK*, as long as the name begins with a letter. The length of the name depends on the system; refer to your system documentation to find out how many letters your names can contain.

After you've saved the lesson material from this chapter, enter **)CLEAR** to get an empty workspace. Then do the exercises on the following pages. You may want to save your work for the exercises in a different place, by entering:

**)SAVE EXERCISE**

Remember, you can get your work back by entering

**)LOAD MYWORK**

or

**)LOAD EXERCISE**

[The name that you put after **)LOAD** will depend on the name you gave the workspace with the **)SAVE** command.]

## SUMMARY

In this chapter, we learned that:

1. In APL, character data is distinguished from numeric data by enclosing the character data in single quotes. Any character (letters, numbers, or special symbols) can be used in character vectors or character matrices.
2. Numbers can be converted to character data using the format function. The format symbol is `⌞`.
3. Character vectors can be joined using catenate `(,)`.
4. Maximum and minimum are functions that compare two numbers and choose the larger (maximum) or smaller (minimum). The maximum symbol is `⌈`, and the minimum symbol is `⌊`.
5. Maximum, minimum, and the arithmetic functions can be combined with a slash `/` to form new functions called reductions. These functions have the same effect as placing a function symbol between each element of a vector or matrix.
6. Maximum, minimum, and the arithmetic functions can also be combined with the “backslash” `\` to form new functions called scans. When used with plus (“plus scan”), the new function gives a running total.
7. You can select elements from a vector or matrix using indexing. Indexing is done with left and right brackets `[` and `]`. For a matrix, both the row and column number must be identified, and these numbers must be separated by a semicolon `;`.
8. You can produce a string of consecutive whole numbers using the count function `⍳`, and you can produce random numbers with the roll function `?`.

## EXERCISES

- A. 1. Assign the characters `↔→↑↓` to a variable named *ARROWS*. Then display the variable.
- ↙ 2. Assign all the special APL characters to a variable called *CHARS* and display it.

- / 3. Create a 3 by 5 character matrix called *NUMBERS*, where the first row contains the word *ONE*, the second contains *TWO*, and the third contains *THREE*. Remember, we form a character matrix like so:

```
ALPH←2 2ρ'ABCD'
ALPH
```

```
AB
CD
```

Also, remember to use spaces where necessary to make each row of the matrix five characters long.

4. Create a character matrix *ALPH* that looks like so:

```
ALPH
```

```
A
B
C
D
E
```

5. Create a character vector by entering:

```
CV←'      '
```

What do you expect its shape to be? Enter  $\rho CV$  to check.

- ✓ 6. Enter the following:

```
A←'CAN' 'T BE'
```

What do you expect its shape to be? Enter  $\rho A$  to check.

- ✓ 7. Write an expression to find the fifth, second, and third elements of the vector *A* you created in Problem 6. All three letters should print together on the same line.

8. Define the following variables:

```
B←'BLUE'  
C←'BELL'  
D←'JINGLE'  
E←'RINGER'
```

What do you expect to see if you enter each of the following?

```
B,C  
D,C  
C,E
```

Now, enter the above expressions and compare the results to your expectations.

9. Enter the following:

```
ALPH←'ABCDEFGHIJKLMNOPQRSTUVWXYZ'  
SYMB←'▲▼|*○'
```

- Write an expression to return the 20th letter of the alphabet.
- Write an expression to return the fourth character in *SYMB*.
- Join *ALPH* and *SYMB* together.
- Take the shape of the two variables joined together.
- Write an expression to print the first 12 letters of the alphabet.
- Write an expression that returns three letters of the alphabet at random.  
(*HINT*: Use the roll function.)

10. Enter the expression:

```
10ρ'★'
```

What did you expect to see?

11. Enter the expression:

```
M←'THE NUMBER IS '
```

Now join *M* together with the number 20 on the same line.

- ✓12. Enter the expressions:

$P1 \leftarrow 'YOU\ HAVE\ SPENT\ '$   
 $P2 \leftarrow 'DOLLARS.'$

Write an expression that will produce the following display:

$YOU\ HAVE\ SPENT\ 10\ DOLLARS.$

(HINT: Use parentheses around the  $\mp 10$  part.)

- ✓B. Enter the following expressions for use with the problems in Exercises B and C.

$A \leftarrow 3\ 1\ 7\ 6$   
 $B \leftarrow 4\ 1\ 2\ 8$   
 $C \leftarrow 2\ 3\ 4\ 2\ 3\ 9\ 7\ 1$   
 $D \leftarrow 2\ 3\ 1\ 3\ 5\ 6\ 8\ 1$

Write down the answers you expect; then check your answers by entering the expressions into the computer.

- |                       |                       |                        |
|-----------------------|-----------------------|------------------------|
| ✓1. $4\ \uparrow A$   | 6. $B\ \uparrow A$    | 11. $4\ \downarrow D$  |
| ✓2. $4\ \downarrow A$ | 7. $A\ \downarrow B$  | 12. $C\ \uparrow C$    |
| 3. $2\ \uparrow B$    | ✓8. $3\ \uparrow C$   | 13. $C\ \uparrow D$    |
| 4. $2\ \downarrow B$  | 9. $3\ \uparrow D$    | ✓14. $C\ \downarrow D$ |
| 5. $A\ \uparrow B$    | 10. $4\ \downarrow C$ |                        |

- C. Use variables  $A$ ,  $B$ ,  $C$ , and  $D$  from Exercise B above for the following problems. Again, write down the answers you expect; then check your answers by entering the expressions in the computer.

- |                   |                      |                                   |
|-------------------|----------------------|-----------------------------------|
| ✓1. $+ / A$       | ✓6. $\downarrow / A$ | 11. $+ / + / C$                   |
| 2. $+ / B$        | 7. $\uparrow / C$    | ✓12. $+ / + / D$                  |
| 3. $+ / C$        | 8. $\downarrow / D$  | 13. $\uparrow / \uparrow / C$     |
| 4. $+ / D$        | ✓9. $+ / [1] C$      | 14. $\downarrow / \downarrow / D$ |
| 5. $\uparrow / B$ | 10. $+ / [1] D$      |                                   |

- D. Assign the following vector of bank transactions to the variable *TRANSACTION*:

```
TRANSACTION←153.17 -20.00 17.00 -35.75
27.55 12.25
```

This is a record of your bank transactions for a month. Positive numbers are deposits; negative numbers are checks or withdrawals. Use this variable in the following exercises.

1. Write an expression to find the third transaction you made this month.
2. Write an expression to find the ending balance for your checking account for this month. (Assume that the beginning balance for the month was \$0.00.)
3. Write an expression that shows the running balance of your account during the month.
4. Write an expression to subtract the last transaction from the first.
5. Your account is a NOW account that earns 5% interest, computed monthly, based on the ending balance. Write an expression that shows how much interest you earned this month.

- RE* E. You have stored your bank deposits and withdrawals for another account in two separate variables.

```
DEPOSITS←120 25 115 375 800.12 22 75
82.72
WITHDRAWALS←35 100 220 175 372.12 15
```

1. Write an APL expression to show how many deposits you made.
2. Write an APL expression to find the ending balance of the account.
3. Write an expression to add a new deposit of \$450 to the end of the vector *DEPOSITS*.

4. Write expressions to find the largest and smallest deposits.
5. Write expressions to find the largest and smallest withdrawals.
6. Write an expression to subtract the largest withdrawal from the largest deposit.
7. Write an expression to pick out the first, third, and fifth deposits.
8. Write an expression to find the first five withdrawals.

✓F. Enter the expression:

`Z←4 4ρ 'HOW CAN THATBE? '`

Be sure to include the blanks as shown.

1. Display Z.
- ✓2. Write an expression to pick out the first three letters of the first row.
3. Write an expression to find the second, third, and fourth letters in the third row.
- ✓4. Write an expression that will cause only the question mark from Z to be printed.
- ✓5. Write an expression that will print the question mark from Z three times.
- ✓6. Write an expression that chooses a letter at random from the first column.
7. Write an expression that chooses a letter at random from the second row.

## Chapter 4

# Selecting and Analyzing Data

---

In this chapter, you will learn about a powerful set of tools to analyze data and select data from vectors and matrices. By the end of this chapter, you will be able to use these tools to answer many questions about your data, such as:

- Did I write any checks over \$20 this month?
- How many?
- What were the amounts of those checks?
- What is the total amount of those checks?

### AN EXPRESSION FOR SELECTING DATA

In Chapter 3, you learned how to do a lot of things with your data, like totaling it up or picking out individual pieces of data. Now we're going to learn how to pick out pieces of the data that meet certain requirements.

Using indexing, you already know how to choose elements of a vector or matrix if you know their positions. But we're also interested in choosing elements that have certain qualities without knowing their positions. For example, with a vector of deposits:

*DEP←85 125 60 45 89*

we might be interested in all the deposits larger than \$80.

The APL expression for finding the deposits larger than 80 is

*(DEP>80)/DEP*  
85 125 89

Deposits less than 80 would be

*(DEP<80)/DEP*  
60 45

Notice that the two expressions are basically the same—all that changed was the “greater than” (>), “less than” (<) part.



Let's take these expressions apart and see how they work. The left-most parts look pretty familiar—they use the “less than” (<) and “greater than” (>) signs we learned in school. These symbols are usually found on the 3 and 7 keys.

Although the left-most parts of the expressions may look familiar to you, their results may surprise you. Enter:

```

      DEP > 80
1 1 0 0 1
      DEP < 80
0 0 1 1 0

```

The 1s and 0s mean “true” and “false,” or “yes” and “no.” If you look at *DEP*, you'll see that the 1s match up with (correspond to) the things we're looking for—the deposits greater than (or less than) \$80.

```

      DEP
85 125 60 45 89
      DEP > 80
1 1 0 0 1

```

### Compression

The 0s and 1s that result from the *DEP > 80* and *DEP < 80* expressions are used by the “slash” (/) to select the numbers we want from the original vector *DEP*. We call this function “compression.” It “keeps” the numbers in *DEP* that correspond to 1s, and “leaves out” the numbers that correspond to 0s. Enter:

```

      1 0 0 0 0 / DEP
85
      0 0 0 0 1 / DEP
89

```

These expressions chose the first and last elements of *DEP*. Now let's enter the 0s and 1s we got by executing *DEP > 80*:

```

      1 1 0 0 1 / DEP
85 125 89

```

And that's that. The only thing we haven't explained is the parentheses in the expression. They're necessary to make sure *DEP > 80* gets executed first, so that compression has the 0s and 1s it needs to work with. By the way,

---

compression needs an *equal* number of numbers on either side of the slash. So if you forgot a 0 or 1 in the examples above, you probably got a *LENGTH ERROR*:

```

      1 0 0 0 / DEP
LENGTH ERROR
      1 0 0 0 / DEP
              ^

```

If you use a “selection expression” (such as *DEP > 8 0*) though, you can never go wrong. It *always* provides as many 0s and 1s as there are numbers in the vector—provided that you remember to put in the parentheses.

### Relational Functions

Our expression also works with  $\leq$  (less than or equal to),  $=$  (equals),  $\geq$  (greater than or equal to), and  $\neq$  (not equal). All of these are called “relational functions,” since they show a relationship. The symbols are usually found on the 4, 5, 6, and 8 keys, respectively. Enter:

```

      ( DEP = 6 0 ) / DEP
60
      ( DEP ≠ 6 0 ) / DEP
85 125 45 89
      ( DEP ≥ 6 0 ) / DEP
85 125 60 89
      ( DEP ≤ 6 0 ) / DEP
60 45

```

### Selecting Character Data

Selection expressions also work with character data, but only using equals (=) and not equal ( $\neq$ ). Enter:

```

      TEXT ← 'THIS SENTENCE HAS NO BLANKS.'
      ( ' ' ≠ TEXT ) / TEXT
THISSENTENCEHASNOBLANKS.

```

In the example above, *' ' ≠ TEXT* produces 0s for blanks and 1s for everything else. The 1s are then used to select all nonblank characters.

```

      ( ' PEAR ' = ' BEAR ' ) / ' BEAR '
EAR
      ST ← ' *** S ** T ** A ** R ** S *** '
      ( ST ≠ ' * ' ) / ST
STARS

```

---

If you have trouble understanding how these expressions work, enter just the part inside the parentheses. The result will be a string of 0s and 1s. Compare the string with the vector that's being compressed and you'll see that the 1s match up with each letter that's "kept" when you execute the entire expression.

No relational functions other than = and ≠ can be used with character data.

```
'PEAR' > 'BEAR'
DOMAIN ERROR
'PEAR' > 'BEAR'
^
```

## COUNTING OCCURRENCES

We might also be interested in the *number* of deposits greater than \$80. The expression `DEP > 80` produces a list of 0s and 1s (1s corresponding to the deposits greater than \$80). The *number* of the deposits greater than \$80 can easily be found by adding the 1s in the list using plus reduction. Enter:

```
+ / DEP > 80
3
```

This approach works equally well with other relational functions.

```
+ / DEP = 60      (number of deposits equal to $60)
1
+ / DEP ≥ 20      (number of deposits greater than or equal to $20)
5
```

We can also use plus reduction in this way to find out how many I's there are in the word MISSISSIPPI:

```
+ / 'I' = 'MISSISSIPPI'
4
```

To find the number of blanks in a sentence, enter:

```
S ← 'IT WAS THE BEST OF TIMES.'
+ / ' ' = S
5
```

## MEETING TWO CONDITIONS

You might also want to select data that meets two requirements. That is, instead of just finding deposits greater than \$80, you might want to find deposits greater than \$80 but less than \$100.

### And

APL provides the function “and” (^) for this purpose. Enter:

```
((DEP>80)^DEP<100)/DEP
85 89
```

In words, we say we want “deposits greater than \$80 *and* less than \$100”; the APL function “and” does this for us. “And” works by comparing 0s and 1s. Only where 1s occur in *both* lists (that is, where *both* conditions are met) does it give us a 1. Enter:

```
      0 1^0 0
0 0      0 1^1 1
0 1      0 0 1 1^0 1 0 1
0 0 0 1
```

Since both  $DEP > 80$  and  $DEP < 100$  produce lists of 0s and 1s, “and” can use these results to find the elements of  $DEP$  that meet both requirements. Enter:

```
      DEP>80
1 1 0 0 1
      DEP<100
1 0 1 1 1
      1 1 0 0 1^1 0 1 1 1
1 0 0 0 1
      DEP
85 125 60 45 89
```

### Or

“Or” is a function related to “and.” It chooses data that meets *either* of two requirements. In words, we say we want “deposits larger than \$120 *or* less than \$75.” The “or” function (v) does this for us. Enter:

```
((DEP>120)vDEP<75)/DEP
125 60 45
```

“Or” also compares 0s and 1s. It produces a 1 if *either* condition is met. Enter:

```

      0 1∨0 0
0 1      0 0 1 1∨0 1 0 1
0 1 1 1
      DEP>120
0 1 0 0 0
      DEP<75
0 0 1 1 0
      0 1 0 0 0∨0 0 1 1 0
0 1 1 1 0
      DEP
85 125 60 45 89

```

## CHECKING ALL OF THE DATA

“And” and “or” can both be combined with a slash to form “and reduction” and “or reduction.” These functions are handy for finding out whether any or all of the data meet certain requirements.

### And Reduction

For example, given a vector containing salary data:

```

      SAL←21000 18000 54000 24000 17000 20000
29500

```

do all of the salaries exceed \$12,000?

```

      ^/SAL>12000
1

```

The answer is yes. Let’s look at this expression in two pieces to see what’s happening. The right-most part finds out whether the elements of *SAL* are greater than 12000:

```

      SAL>12000
1 1 1 1 1 1 1

```

Then the “and reduction” (^/) part checks to see whether *ALL* of the salaries are greater than \$12,000 (that is, whether the results of *SAL>12000* are *all* 1s):

```

      ^/1 1 1 1 1 1 1
1

```

Let's look at *SAL* again:

*SAL*  
21000 18000 54000 24000 17000 20000 29500

If we ask whether every salary is larger than \$17,000, we get one 0 in the result:

*SAL > 17000*  
1 1 1 1 0 1 1

If we check this result with “and reduction,” we get a 0 since at least one of the salaries is not greater than \$17,000.

*^ / 1 1 1 1 0 1 1*  
0

In other words, just one false result makes the entire result false.

### Or Reduction

We can use “or reduction” to find out if *any* of the data meets a requirement. For example, does anyone in our group have a salary larger than \$50,000?

*v / SAL > 50000*  
1

Going back to our deposits example, are any of our deposits larger than \$150?

*v / DEP > 150*  
0  
*DEP*  
85 125 60 45 89

### SELECTING DATA FROM MATRICES

Compression works a bit differently with matrices. Remember that a matrix has two dimensions (or coordinates), meaning that an operation can be carried out either along the rows dimension or along the columns dimension. With that in mind, let's give it a try and see what happens.

```

      MAT←4 4ρ16
      MAT
1  2  3  4
5  6  7  8
9 10 11 12
13 14 15 16
      0 1 0 1/MAT
2  4
6  8
10 12
14 16

```

compressing  
columns

It's clear that APL compressed *MAT* by removing entire columns of numbers—to be precise, the first and third columns. That is, it compressed the *second* dimension of *MAT* (the columns), as we saw before with reduction in Chapter 3.

compressing  
rows

To get APL to compress the first dimension (the rows), we do the same thing we did for reduction—add [1] to the expression.

```

      0 1 0 1/[1]MAT
5  6  7  8
13 14 15 16

```

The first and third rows were compressed out, exactly as we wanted. This also works with character matrices:

```

      CHARMAT←3 3ρ'CATRATBAT'
      CHARMAT
CAT
RAT
BAT
      0 0 1/[1]CHARMAT
BAT
      0 0 1/CHARMAT
T
T
T

```

length  
restrictions for  
compressing  
matrices

Notice that the left argument to compress must be a vector. The length of the vector must match the length of the dimension that the vector is compressing. That is, if you're compressing the columns' dimension, and there are five columns, then the compression vector should contain five 0s and 1s.

using vectors  
as keys for  
compressing  
matrices

We can use a vector as a "key" to a matrix, and do our searching on the vector. Then we use the result of the search to compress the matrix. Let's look at an example.

Suppose we have a vector of salesman numbers for four salesmen:

```
SALESMEN←101 112 126 128
```

We also have a matrix of sales data:

```
SALES←4 3p1075 3023 4010 2075 750
1189 2211 3117 2288 2779 1077 1928
SALES
1075 3023 4010
2075 750 1189
2211 3117 2288
2779 1077 1928
```

We've set up the matrix so that each row corresponds to one salesman's revenues for the first three months of the year. In other words, row 1 corresponds to salesman 101, row 2 corresponds to salesman 112, and so on. Now, if we want to find out where salesman 101 is located in the vector *SALESMEN*, we enter:

```
SALESMEN=101
1 0 0 0
```

And if we want to look at his sales, we enter:

```
1 0 0 0/[1]SALES
1075 3023 4010
```

which is the same as entering:

```
(SALESMEN=101)/[1]SALES
1075 3023 4010
```



Since there's a perfect match between the rows of *SALES* and the elements of *SALESMEN*, we can compress *SALES* using a vector of 0s and 1s that we get from *SALESMEN* using our selection expression. Let's follow through with our example. Create a character matrix containing our salesmen's addresses so that it looks like so:

```

      ADDRESS←4 16ρ'21 ELM STREET 5107
MAIN STREET52 NORTH ROAD 1717 IOWA
AVENUE'
      ADDRESS
21 ELM STREET
5107 MAIN STREET
52 NORTH ROAD
1717 IOWA AVENUE

```

We can find the address for salesman 101 using the same formula:

```

      (SALESMEN=101)/[1]ADDRESS
21 ELM STREET

```

And if we have the salesmen's ages stored in *AGES*:

```

      AGES←39 28 47 35

```

we can find salesman 101's age the same way.

```

      (SALESMEN=101)/AGES
39

```

We could go on and on like this, but you get the point—*SALESMEN* is the key to all the other stored data.

Let's look at one more aspect of this. Remember that the columns of *SALES* represent sales in the first, second, and third months. If we store the numbers of the months in a variable *MONTH*:

```

      MONTH←1 2 3

```

we can select the sales data for a particular month:

```

      (MONTH=3)/SALES
4010
1189
2288
1928
      SALES
1075 3023 4010
2075  750 1189
2211 3117 2288
2779 1077 1928

```

Putting the two pieces together, we can find the sales for salesman 101 during month 3:

```

      (MONTH=3)/(SALESMEN=101)/[1]SALES
4010

```

(SALESMEN=101)/[1]SALES picks out the row with salesman 101's sales, and MONTH=3 selects the third number in the row.

Whew! That's probably enough for one chapter. Time to store our work—with the command )SAVE MYWORK—and move on to some exercises.

## SUMMARY

1. Variations of the expression (DEP>80)/DEP can be used to select pieces of data that meet certain requirements. Expressions of this form combine a relational function (<, ≤, =, ≥, >, ≠) with the compression function (/).
2. Variations of the expression +/DEP>80 can be used to determine how many items meet a certain requirement.
3. Variations of the expression ((DEP>80)^DEP<100)/DEP can be used to select data items that meet *two* requirements. This expression uses the “and” function (^).
4. Variations of the expression ((DEP>120)∨DEP<75)/DEP can be used to select data items that meet *either* of two conditions. This expression uses the “or” function (∨).

5. Variations of the expression  $\wedge / SAL > 17000$  tell us whether *all* of the data meet a certain requirement. This expression uses the “and reduction” function ( $\wedge /$ ).
6. Variations of the expression  $\vee / SAL > 50000$  tell us whether *any* of the data items meet a certain requirement. This expression uses the “or reduction” function ( $\vee /$ ).
7. Vectors can be used as “keys” to select data from matrices or from other vectors, using the selection expressions described in points 1-4.

## EXERCISES

1. Write an APL expression to count the number of Es in TENNESSEE.

2. Enter the vector of bank transactions *TRANSACT*.

```
TRANSACT←180 -25.25 -40.89 87.12
237.25 -127.27 39.45
```

Write an expression to count the number of withdrawals (negative numbers).

3. For the vector *TRANSACT* shown in Problem 2, write an expression to count the deposits (positive numbers).
4. Write an APL expression to count the number of blanks in the following sentence:

```
S←'TO BE OR NOT TO BE, THAT IS THE
QUESTION'
```

5. Enter the following vector of sales figures:

```
SALES←757 212 358 821 375 115 837
227 706
```

Write an expression to count the number of sales that are less than or equal to \$375.

- J B. Use the following variable, which contains the account balances for our 10 main customers, to work the exercises in this section.

AW

**ACCOUNTS**+135 27 -15 12 -25 -57 240 172  
18 29

Write APL expressions to answer the questions listed below.

1. Do all the accounts have *positive* balances?
2. How many have negative balances?
3. How many have positive balances?
4. Are there any accounts greater than \$200? How many?
5. Are there any accounts between \$120 and \$175? How many?
- J J 6. What is the grand total of the accounts with negative balances? (*HINT*: Use compression to find the negative balances, then add them up with plus reduction.)
- J J 7. What is the grand total of the accounts with positive balances?

- C. Use the following price information to work the exercises in this section.

**PRICES**+1.79 5.21 10.77 30.29 18.95  
19.95 3.07 2.15

1. Write an expression to select the prices larger than \$15.00.
  2. Write an expression to select the prices less than or equal to \$3.07.
  3. Write an expression to select any prices equal to \$17.75.
  4. Write an expression to select prices larger than \$10 but smaller than \$20.
  5. Write an expression to select numbers either less than \$2 or greater than \$30.
-

D. Use the following variables to work the exercises in this section.

Class

A ← 3 3p19  
A

1 2 3  
4 5 6  
7 8 9

CM ← 3 3p'MOEJOETOE'  
CM

MOE  
JOE  
TOE

1. 0 0 1/A : 3 4 0E  
2. 0 1 1/CM ~ MOE  
3. 1 0 0/[1]CM = MOE

4. 0 1 0/[1]A = 456  
5. 0 0 1/0 1 0/[1]A = 6  
6. 0 0 0 1/A  
LENGTH ERROR

E. Use the variables shown below to work the exercises in this section.

All

CUSTNO ← 1 2 3 4  
MONTHS ← 10 11 12  
CADDRESS ← 4 16p'55 WESTLAKE DR. 101  
JEFFREY LN. 23 REPUBLIC AVE. 27 MEMORIAL DR. '  
CORDERS ← 4 3p73 412 811 27 84 11 72 35  
99 107 78 23

CUSTNO and MONTHS are keys to the other data. The customer numbers (CUSTNO) correspond to the rows of CADDRESS and CORDERS. The months correspond to the columns of CORDERS (that is, CORDERS contains customer orders for three months: October, November, December).

1. Write an expression to display customer 3's orders during this period.
2. Write an expression to display customer 4's address.
3. Write an expression to display all the orders for November.
4. Write an expression to display customer 1's orders in December.

## Chapter 5

# Writing Programs in APL

---

In this chapter, you will learn:

- how to enter function definition mode and create your own programs
- how to run your programs
- how to make corrections in your programs
- how to use character data in programs
- about the six different classes of APL programs

### note

The programs appearing in Chapters 5 through 9, including those appearing in the exercises, are stored in the workspace *LESSONS* supplied with all APL\*PLUS Systems. If you ever get tired of entering a program or you just can't seem to get it right, you can always copy the version stored in *LESSONS*. Just enter:

```
)COPY LESSONS pgmname
```

*Pgmname* is the name of the program you want to copy. For example:

```
)COPY LESSONS AVG
```

If you're using a personal computer APL system and you receive the message *WS NOT FOUND*, you should try inserting one of the other disks supplied with your system in your primary disk drive. Then enter the command again.

The code (that is, the APL programs) for the workspace is also listed in Appendix D. If your APL system does not have the *LESSONS* workspace, you can always type the programs in using the code in Appendix D as a guide.

By now, you're probably tired of typing in the same lines every time you want to do something. You might have thought at least once, "There *must* be an easier way to do this!" Well, it turns out that you can store APL expressions, much as you can store data.

In this chapter, you will learn how to store generally useful expressions. In computer jargon, this is called “writing programs.” In APL literature, it’s usually called “defining functions” or “writing user-defined functions,” but as we said in Chapter 1, the words “program” and “function” mean the same thing in APL.

## WHY WRITE PROGRAMS?

Why would you want to write your own program? One reason has already been given—you’d like to avoid typing in the same lines over and over. Another reason is that programs can store *many* lines of APL, allowing them to perform complicated tasks, as we saw in Chapter 1. Also, programs can “call” other programs and thus perform even more complicated tasks. Finally, programs can be stored in workspaces and used again and again for similar work.

## AN EXAMPLE OF AN APL PROGRAM

One example of using a program to avoid a lot of extra typing is taking an average. The general APL expression for this is:

$$(+ / VECTOR) \div \rho VECTOR$$

If you think about this expression for a minute, you’ll see how it works. To use an example, if you want to average three numbers, you add up all three, then divide by three (that is, divide by the number of numbers). That is,

$$\begin{array}{c} (2 + 3 + 4) \div 3 \end{array}$$

In the general case, we add up all the numbers (+ /) then divide by the number of numbers (shape,  $\rho$ ). Let’s try our expression:

$$\begin{array}{c} V \leftarrow 1 \ 2 \ 3 \ 4 \ 5 \\ (+ / V) \div \rho V \end{array}$$

Now, this is a lot to type, and what’s more, it’s a lot to remember. It might be nice to have a function (like plus or divide) called *AVG*, so that we could enter

$$AVG \ V$$

and get the result we want. In other words, we want something that *we* define, but that acts like a built-in APL function. This is the basic idea of a user-defined function. We create an expression to perform a particular task, give the

expression a name, and then use the named expression just like APL functions such as multiply and plus reduction. (This is why programs are usually called functions in APL.)

## DEFINING A FUNCTION

To create an APL function, we must enter “function definition mode.” This is different from “immediate execution mode” because the system *doesn't* execute the statements as we type them in or give us any answers. Instead, it stores the lines under the name we give so that they can be used later.

### note

Most APL systems have a full-screen text editor that you can use to create and modify functions. Since these facilities differ from machine to machine, we strongly recommend that you read the documentation provided with your system to learn how to use the editor, or ask an experienced user.

In addition to the full-screen editor, most APL systems also have a line-oriented editor called the del editor. We'll explain here how to use the del editor to create and modify short functions. Be sure to learn how to use your system's full-screen editor, however, to make entering the functions in later chapters easier.

### entering function definition mode

To enter function definition mode, type a “del” (▼), which is usually found on the G key. The del should be followed by the name of the function and the name of any data it will use. Enter:

```
▼AVG DATA  
[ 1 ]
```

### function header

Be sure to leave a space between *AVG* and *DATA*. Together, these two items are called the “function header.”

Notice that the system immediately types a 1 surrounded by brackets. This indicates (1) that we are in function definition mode, and (2) that the computer is waiting for us to enter the first line of our function. So let's give it something.

### entering lines of a function

Enter, on the same line with the [ 1 ]:

```
[ 1 ]      ( + / DATA ) ÷ ρ DATA  
[ 2 ]
```



The system printed a 2 in brackets, because it's now waiting on the *second* line of our function. We've already typed in all we want to though, so let's exit from function definition mode by typing another del on the line with the [2]:

```
[2]      ▽
```

note

Some APL systems may require a different sequence to exit from function definition mode. (For example, you must use a pull-down menu in the APL\*PLUS System for the Macintosh.) Check your system documentation if this example does not work for you.

By now, you should have a display like the following one in front of you.

```
      ▽AVG DATA
[1]      (+ / DATA) ÷ ρ DATA
[2]      ▽
```

If you do, congratulations! You've just created your first function. If your function *doesn't* look exactly like this, it may not work. But don't worry—we'll learn how to correct typing errors in a minute.

## EXECUTING A DEFINED FUNCTION

Test your function with some numbers you know the average of:

```
      AVG 90 100
95
      AVG 100 200 300
200
```

Then try some tougher ones:

```
      AVG 37 74 29
46.66666667
      AVG 1.7 2.1 3.2
2.333333333
```

Success! At least it was for those of you who typed the function in correctly. For those of you who didn't, it's time to learn how to correct a function definition. Those of you who got it right the first time will need to read this section too, since you're bound to make a mistake sooner or later.

## CORRECTING MISTAKES

We call the process of correcting mistakes in a function “editing a function.” To edit a function, you need to get back into function definition mode by typing a  $\nabla$  and the function’s name:

```
       $\nabla$ AVG
[2]
```

## Replacing an Entire Line

Notice that you *shouldn’t* type in the name of the data used by the function (*DATA*, in this case) in order to edit it. Also notice that the computer printed a [2]. The system assumes that we want to add lines to the *end* of the existing function definition. To direct the computer’s attention elsewhere, simply enter the number of the line you want to go to (in this case, line [1]):

```
[2]      [1]
[1]
```

The system responds with a [1], indicating that it’s waiting for you to enter a *new* line [1]. This new line [1] will replace the old one. Enter:

```
[1]      (+ / DATA)  $\div$   $\rho$ DATA
[2]
```

(As you can see, it doesn’t matter if you enter the exact same thing for the new line [1].) Now that it’s got the new line, the system is ready to move on to the next line. We’re finished though, so we enter a del.

```
[2]       $\nabla$ 
```

You should now have a display similar to the following in front of you:

```
       $\nabla$ AVG
[2]      [1]
[1]      (+ / DATA)  $\div$   $\rho$ DATA
[2]       $\nabla$ 
```

Test your function again to make sure it works:

```
      AVG 25 50 75
50
```

**Correcting Part  
of a Line**

We've learned how to edit a function by replacing an entire line with a new one. In some cases, we might want to change only part of the line, particularly if the expression is a long one. Let's create a new function to demonstrate this editing technique. Enter:

```
▽ERASEBLANK S
[1]
```

This function will delete all the blanks in a sentence. Let's enter the working part of the function, but let's make a mistake in doing it:

```
[1]    (' '≠S)/W
[2]    ▽
```

Notice that the *W* at the end of the expression should have been an *S*. If we try to run this function with the error in it, we'll get an error message:

```
ERASEBLANK 'SO IT IS'
VALUE ERROR
ERASEBLANK[1] (' '≠S)/W
                        ^
```

We get a *VALUE ERROR* because *W* doesn't have a value, some data, stored in it (we haven't defined a *W*). Now that APL has identified the mistake in the function, we want to get back into function definition mode and correct it.

It's a good idea to enter the command:

```
)SIC
```

*Clear state indication  
if error causes  
suspension*

before you edit a function that has run into a problem. "Stopped" functions are said to be "suspended," and the system keeps track of them in something called a "stack" (that is, a stack of things for the system to do). The command *)SIC* clears out (resets) the stack, so that the system doesn't get confused when we try to run the function again later (after we've fixed it). If you *don't* enter *)SIC*, you may get a *DEFN ERROR* (definition error) when you try to run the corrected function.

Now let's correct our function. Enter:

```

      ▽ERASEBLANK
[2]   [1□0]

```

The system is ready to accept line [2], but we tell it to go to line [1], print it, and leave us at the end of the line. The only new character here is the quad (□), usually found on the L key, which separates the desired line number (1) from the desired position on the line (0, meaning "end of line"). The system responds:

```

[1]   ( ' ' ≠ S ) / W

```

The cursor is now located at the right of the line. To make our correction, backspace one space (using the BACKSPACE or UNTYPE key). This causes the letter *W* to be erased. We can now type in the correct letter:

```

[1]   ( ' ' ≠ S ) / S
[2]   ▽

```

The system was ready for line [2], but we were done, so we entered a del.

Test the function:

```

      ERASEBLANK 'YOU CAN SAY THAT AGAIN'
YOU CANSAYTHATAGAIN
      ERASEBLANK '* * * OOPS'
***OOPS

```

#### note

There are many techniques for correcting lines in a function definition. We have introduced only the simplest methods here. If you are interested in other methods, you should refer to the documentation provided with your particular APL system.

#### Displaying a Function Definition

Another thing we might want to do with a function is display it. This is especially useful when the function has many lines. Let's create a function with a number of lines that tells us about our bank account. In particular, it tells us the largest deposit, the smallest number (which, if negative, is the largest withdrawal), and the present balance. We'll call the data *V* (for "vector"). Enter:

```

      ▼ACCOUNT V
[1]   ⍒/V
[2]   ⍒/V
[3]   +/V
[4]   ▼

```

Let's test our new function:

```

      ACCOUNT 70 130 35 -21 -12 27
130
-21
229
      A←30 -22 295 7 -190 150 -55
      ACCOUNT A
295
-190
215

```

If we came back the next day, we might not remember what these three numbers are, so we would want to list the contents of the function to refresh our memories. Enter function definition mode:

```

      ▼ACCOUNT
[4]

```

Now type a [□] (again, using the quad symbol) to list the function:

```

[4]   [□]
[0]   ACCOUNT V
[1]   ⍒/V
[2]   ⍒/V
[3]   +/V
[4]

```

The computer lists everything in the function, including its first line (the name line), then waits for line [4]. Since we only want to list the function, we exit function definition mode by entering a del:

```

[4]   ▼

```

### Inserting a Line

Another thing you might like to do with a function is insert a line. Let's say that we really meant to include a line for calculating the running balance in our function *ACCOUNT*. We wanted this line to come right after line [2]. Enter function definition mode:

```
      ▼ACCOUNT
[4]
```

If you now enter [2.5], the system will get ready to accept a *new* line to insert between lines [2] and [3]:

```
[4]      [2.5]
[2.5]
```

We can now enter the statement for taking a running balance:

```
[2.5]      +\V
[2.6]
```

Now the system is ready to insert a *second* line between lines [2.5] and [3]. We have nothing else to add, but we *would* like to see whether the line got inserted, so we enter the command for displaying the function, [□]:

```
[2.6]      [□]
[0]      ACCOUNT V
[1]      ⍒/V
[2]      ⍒/V
[2.5]      +\V
[3]      +/V
[4]
```

Satisfied that the line has been inserted, we exit function definition mode:

```
[4]      ▼
```

Let's test the new version of the function:

```
ACCOUNT A
295
-190
30 8 303 310 120 270 215
215
```

The display now includes a fourth item, the running balance.

Just for curiosity's sake, let's display the function once more:

```
▽ACCOUNT
[5]  [ ]
[0]  ACCOUNT V
[1]  ⍒/V
[2]  ⍒/V
[3]  +\V
[4]  +/V
[5]
```

Surprised? Line [2 . 5] has become line [3]. The system automatically renumbers the function lines when we exit function definition mode. This makes it easier to keep track of line numbers when we begin writing and editing longer functions.

### Erasing a Line

While we're still in function definition mode, let's look at another thing we can do with function lines—delete them. We really don't need line [4], since the running balance shows the ending balance anyway. Enter:

```
[5]  [~4]
[4]
```

(The tilde character, ~, is usually found on the T key.) On some computers, you would use a delta (Δ) instead of a tilde:

```
[5]  [Δ4]
[4]
```

Let's list the function to make sure line [ 4 ] has been deleted:

```
[4]      [0]
[0]      ACCOUNT V
[1]      /V
[2]      L/V
[3]      +\V
[4]
```

Then exit function definition mode:

[4] ▽

## SUMMARY OF EDITING COMMANDS

A summary of the five editing commands we've learned is given in the table that follows.

| Command                                                                               | Description                                                                                                                                                                                                                                |
|---------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| [n]                                                                                   | Send the computer to line [n]. It waits for a new line [n] to replace old line [n].                                                                                                                                                        |
| [n]  | <del>Send the computer to line [n], print the line, then have it wait at the end of the line. Use the UNTYPE or BACKSPACE key to edit the line.</del>  |
| [ ]                                                                                   | Display the entire function.                                                                                                                                                                                                               |
| [n.m]                                                                                 | Insert a line between two existing lines. For example, [1.5] (or [1.1] or [1.7]) inserts a line between existing lines [1] and [2].                                                                                                        |
| [~n] or [Δn]                                                                          | Delete line [n].                                                                                                                                                                                                                           |

also

$\{a-b \square\}$  display } lines  $a-b$   
 $\{\Delta a-b\}$  delete }

 ~~$\{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z\}$~~



**ADDING  
CHARACTER  
DATA TO  
FUNCTIONS**

It would be nice if *AVG* explained what its answer means, like this:

```
      AVG 2 3 4  
THE AVERAGE IS 3
```

This would be really helpful with the function *ACCOUNT*, since it calculates so many different answers.

Using character data in a function is as simple as using character data in immediate execution mode. Remember, we simply surround characters with single quotes and the computer “throws it back” at us:

```
      'YOU DON'T SAY'  
YOU DON'T SAY
```

The only exception is the quote itself, which must be doubled, as the example shows.

The second thing to remember is that numbers can be converted to characters using the format function and then catenated to other character data. Format uses the symbol  $\mathbb{F}$ , and catenate uses the comma symbol. For example:

```
      'THE AVERAGE IS ' ,  $\mathbb{F}$ 3  
THE AVERAGE IS 3
```

We can also put an APL expression on the right-hand side of format:

```
      'THE SUM IS ' ,  $\mathbb{F}$ 2+3  
THE SUM IS 5
```

APL does the arithmetic first, then turns the answer (5) into character data, then catenates it to the sentence. This is the form we will use to get explanatory sentences for the answers in our functions.

Let's add some explanatory notes to our *ACCOUNT* function. Rather than edit each line of *ACCOUNT*, let's create a new version called *ACCOUNT2*. Enter:

```
∇ACCOUNT2 DATA
[1] 'THE LARGEST DEPOSIT IS ',⌈/DATA
[2] 'THE LARGEST WITHDRAWAL IS ',⌈/DATA
[3] 'THE RUNNING BALANCE IS ',⌈+ \DATA
[4] ∇
```

You may want to list your program to make sure everything's okay. In particular, make sure you have a quote at the beginning and end of each piece of text.

Test the function:

```
ACCOUNT2 78 122 -23 -11 -55 67
THE LARGEST DEPOSIT IS 122
THE LARGEST WITHDRAWAL IS -55
THE RUNNING BALANCE IS 78 200 177 166 111 178
```

If the system reports any errors, enter *)SIC*, get back into function definition mode, go to the line where the system showed the error, and correct it. You may find that once you've corrected an error in line [1], the system will find another error in line [2] or [3]. But once you get your function to look exactly like the one above, it'll work perfectly.

## DIFFERENT FUNCTION FORMS

So far, we've created a lot of different functions, but all had the same basic form:

*function name data*

## Functions with One Argument

For example:

```
AVG DATA
ERASEBLANK S
```

We mentioned that this is similar to the functions plus reduction or shape:

```
+ / A
ρ A
```

**Functions with  
Two Arguments**

All of the functions shown use only one argument. Many functions, however, use two arguments (that is, take data on both sides). For example,

```
      6 ÷ 2
3
0 1 / 'AZ'
Z
```

When creating your own functions, you may want them to take data on both sides. Consider the function *PERIMETER*:

```
      5 PERIMETER 6
22
```

This function takes the length of two sides of a rectangle and finds the perimeter. Remember from geometry that the perimeter of a rectangle is the sum of the lengths of the sides.

It is as easy to build functions with two arguments as it was to create functions with one argument. Enter:

```
      ∇WIDTH PERIMETER LENGTH
[1]
```

Simply by entering three names separated by blanks, you have told the computer that:

1. The function's name is *PERIMETER*.
2. One piece of data (the left argument) is called *WIDTH*.
3. A second piece of data (the right argument) is called *LENGTH*.

Now let's enter the working part of the function. You can find the perimeter by adding the lengths of the two sides and multiplying by 2, so:

```
[1]      2×WIDTH+LENGTH
[2]      ∇
```

Now let's test it:

```
10 PERIMETER 20
60
5 PERIMETER 8
26
```

Encouraged by your experience with *PERIMETER*, you may feel tempted to define a function with *three* different pieces of data. For example, you might want a function using width, length, and height to find the volume of a cube. But if you enter:

```
    ∇WIDTH VOLUME LENGTH HEIGHT
DEFN ERROR
```

APL gives you an error message and returns you to immediate execution mode. The reason is simple. If you enter three names, APL assumes that the one in the middle is the function name, and the two names on either side are data names. Similarly, if you enter two names (such as *AVG V* or *ERASEBLANK S*), it assumes that the function name is the name on the left. But if you enter more than three names, APL has no idea which name is which. So such functions are not allowed in APL. This is really no problem, though, because we can use more than one number in the data for our functions, as we saw with *AVG* and *ACCOUNT*. For a *VOLUME* function, you'd probably want to define it with one argument:

```
VOLUME LWH
```

where *LWH* would be a three-element vector containing the length, width, and height.

### Functions with No Arguments

There is a third type of function that has *no* arguments. We can define a function *DIE*, which "rolls a die" and displays the number:

```
    DIE
1
    DIE
2
    DIE
4
```

This function uses the built-in APL function called roll (?) that you learned about in Chapter 3. Let's define it:

```
▽DIE
[1] ?6
[2] ▽
```

Test it:

```
5 DIE
4 DIE
```

(Your results will be different, but the function always produces a random number between 1 and 6.)

You may think that functions that take no data would have few uses, but they are very useful for doing things like printing a report or controlling files. We'll learn more about such uses in Chapters 8 and 9.

## Explicit Results

One important characteristic of the built-in APL functions is that they return "explicit results." This means that one APL function provides numbers that can be used by other APL functions for more work. For example, in the expression

```
5+2 3×1×2
7 11
```

the `2 3×1×2` returns the explicit result `2 6`, which is then used by the `5+` part to get the answer `7 11`. We never see the `2 6` result, because it's automatically used by the `5+` part of the expression.

We've said that our user-defined functions act as "stand-ins" for built-in functions, so we should expect them to work in a similar fashion. In other words, we should be able to enter

```
2+AVG 9 10 11
```

and get an answer. But what happens if we try it?

---

```

      2+AVG 9 10 11
10
VALUE ERROR
      2+AVG 9 10 11
      ^

```

The problem is that we haven't defined *AVG* to return an explicit result. The *AVG* function displays the answer (10 in this case) and then promptly "forgets" it. The answer is not stored anywhere, so APL has nothing to add 2 to. And we get a *VALUE ERROR* message.

### Defining Functions with Explicit Results

The *VALUE ERROR* problem described above is an easy problem to fix. All we have to do is tell the function to store the result somewhere. Let's define a new version of *AVG* to demonstrate this technique. Enter:

```

      ▽R←AVG2 A
[1]

```

Notice what's new here. We're saying that the result of the function will be in a variable *R* (for "result") at the end of the function. We've also called the argument *A* this time, just to point out that we can call the arguments whatever we want.

Let's enter the rest of the function:

```

[1]      R←(+ / A)÷ρA
[2]      ▽

```

Again, notice what's new—we've assigned the result of the APL expression to the variable *R*. If you want the function to have an explicit result, you must (1) name the result variable in the name line of the function (the "header"), and (2) assign the result data to that result variable in the body of the function. The name of the result variable must be the same in both places.

Let's try our experiment again, using our new function:

```

      2+AVG2 9 10 11
12

```

Eureka! So it's possible to build functions that are used the same as built-in APL functions. This means that we can use the results of one function in another function:

```
2  PERIMETER AVG2 9 10 11
24
2  PERIMETER AVG2 1 2 3
8
```

In the first example, *AVG2* gives the result 10, and then *2 PERIMETER 10* gives the answer 24. One function (*AVG2*) gives a result, which is then passed on to the next function to give the “final” answer. Only the final answer is displayed by the computer.

Since our new *AVG2* function returns an explicit result, we can capture the result in a variable:

```
ANS←AVG2 9 10 11
ANS
10
```

This was impossible with our original function *AVG*:

```
ANS←AVG 9 10 11
10
VALUE ERROR
ANS←AVG 9 10 11
      ^
```

for the same reasons we explained above.

Let's make alternative versions of two other functions:

```
▽R←WIDTH PERIM LENGTH
[1] R←2×WIDTH+LENGTH
[2] ▽

▽ANS←DIE2
[1] ANS←?6
[2] ▽
```

Now we can use these functions as arguments to each other. For instance, we can “roll two dice” and show the result:

```

        DIE2 + DIE2
6
        DIE2 + DIE2
9

```

We can also assign their results to variables.

```

        Q ← 4 PERIM 6
        Q
20

```

You can see that it’s useful to have functions that return explicit results, so you’ll probably want to define most of your functions in this way.

## Function Syntax

The following table summarizes the six different forms of functions we’ve learned in this chapter. These are all the forms of functions that are defined in APL. Another word for the form of a function is its “syntax.” This word isn’t entirely new to you, since you’ve seen *SYNTAX ERROR* messages before. (A *SYNTAX ERROR* means that you’ve mixed up the positions of the function and the data—like *V + /* or *V ρ.*) The chart shows the syntax of each function type (using *FN* as the function name, *R* as the result, and *A* and *B* as argument names) and also shows examples of each type (the functions we created in this chapter).

|                      | No Explicit Result                             | Explicit Result                                    |
|----------------------|------------------------------------------------|----------------------------------------------------|
| <b>No Arguments</b>  | <i>FN</i><br><i>DIE</i>                        | <i>R ← FN</i><br><i>R ← DIE2</i>                   |
| <b>One Argument</b>  | <i>FN A</i><br><i>AVG DATA</i>                 | <i>R ← FN A</i><br><i>R ← AVG2 A</i>               |
| <b>Two Arguments</b> | <i>A FN B</i><br><i>WIDTH PERIMETER LENGTH</i> | <i>R ← A FN B</i><br><i>R ← WIDTH PERIM LENGTH</i> |



Before you go on to the exercises, you may want to store your functions and variables. Functions and variables are stored using the same command:

**`)SAVE MYWORK`**

Remember, you can list the functions in your workspace using the command `)FNS`. Remember too, that all the programs shown in this chapter and in the rest of the book are in the workspace *LESSONS*.

## SUMMARY

1. It is useful to create your own functions, because they:
  - save typing
  - are easier to remember
  - can be stored for future work
  - can be used to take the place of many APL statements
  - can be used to build systems
2. You enter function definition mode by entering a del (`▽`) and leave it by entering another del.
3. A function consists of a “header” (which gives the function its name and identifies its data) and a “body” (which does the work).
4. Functions can be edited using the commands summarized on page 74.
5. Character data can be used in a function just as it is used in immediate execution mode.
6. Functions can have either no arguments, one argument, or two arguments.
7. To use the results of a function in other calculations, you must define the function to return an explicit result.

## EXERCISES

- ✓ A. 1. Create the function shown below. Note that this is just a test program with which to practice your editing skills. If you try to execute it, you'll get a *VALUE ERROR*.

f.c.t.  
practice  
only

```

          ▼NUMBERS
[1]      ONE
[2]      TWO
[3]      THREE
[4]      FOUR
[5]      FIVE
[6]      ▼
    
```

2. Enter function definition mode and replace line [2] of *NUMBERS* with the word *TWO*.
  3. Replace line [4] with the word *FORE*.
  4. Using the editing command [*n*□0], change line [3] to read *TREE*.
  5. Add a line between lines [1] and [2] containing the word *TWO*.
  6. Delete line [5].
  7. List the function.
  8. Change the name of the function to *COUNT*. (Use the command [0] to do this—line [0] is the function header.)
  9. List the function.
  10. Exit the function.
- B. 1. Write (on a piece of paper) the header for a function called *ADD* that will have two arguments and will return an explicit result. The function should work as shown below.

```

          4 ADD 5
9
          ANSWER←4 ADD 2
          ANSWER
6
    
```

2. Write the body of the function *ADD*.

3. Create the function *ADD* on the system and test it.
4. Write (on a piece of paper) the header for a function *RUNTOT* that takes one argument and gives a running total of the data in the argument. The function should *not* return an explicit result.
5. Write the body of the function *RUNTOT*.
6. Create the function *RUNTOT* on the system and test it. Use the numbers 10, 20, 30, 40.

- C. 1. Create a function called *HI*, which works as follows:

```
HI
HELLO. MY NAME'S APL.
WHAT'S YOURS?
```

The function does no arithmetic; it merely prints the message.

2. Create a function called *HALF* that returns one-half of the number (or numbers) you give it. What's one-half of 3.14159?
3. If you didn't do it the first time, redefine *HALF* so that it returns an explicit result.
4. Use the function *HALF* as part of an APL expression to find the sum of 10 and one-half of 3.14159.
5. Define a function called *ROWTOT* that returns the row totals for a matrix. For example,

```
MAT←3 2ρ16
MAT
1 2
3 4
5 6
ROWTOT MAT
3 7 11
```

6. Define a function called *COLTOT* that returns column totals. For example,

```
COLTOT MAT
9 12
```

7. Try your *ROWTOT* and *COLTOT* functions on this matrix:

```
MAT2←3 4ρ10+112
MAT2
11 12 13 14
15 16 17 18
19 20 21 22
```

8. Use your *COLTOT* function inside a new function *RPT*, which displays a simple report based on a matrix. The report should:

- print the data (the matrix)
- print a line of underlines (hyphens)
- print the column totals (use *COLTOT MAT2* for this part)

The function should produce the following display:

```
RPT MAT2
11 12 13 14
15 16 17 18
19 20 21 22
-----
45 48 51 54
```

(Notice that the columns aren't perfectly lined up. We'll learn how to print fancier reports in Chapter 6, with nicely lined up columns, dollar signs, and other things.)

- D. 1. Write a function *REPEAT* that has the following effect:

```

      REPEAT 'ANN'
ANN ANN ANN ANN ANN
      REPEAT 'JO'
JO JO JO JO JO

```

(*HINT*: Use catenation.)

2. You can find the square of a number by multiplying it by itself:

```

      3 × 3
9

```

Define a function called *SQUARE* that produces the following display:

```

      SQUARE 9
THE NUMBER SQUARED IS 81

```

What is the square of 247?

3. Define a function *CHOOSE* that has the following effect:

```

      8 7 12 73 55 CHOOSE 3
12
      101 25 14 47 CHOOSE 2
25

```

That is, it “chooses” a number from the list on the left based on the position specified on the right. The function should also work if you use it like so:

```

      NOS←8 76 43 21 23 6
      NOS CHOOSE 4
21

```

(*HINT*: Use indexing.)

4. Create a function called *OF* which has two arguments and works as follows:

```

      20  OF  '*'
*****
      4   OF  '□'
□□□□

```

In other words, it gives you as many *of* the things on the right as you ask for with the number on the left. You should also be able to enter:

```

      2 3  OF  '?'
?? ?
?? ?

```

(This should give you a clue as to how the function works.)

5. “Histogram” is another word for a bar chart, which compares numbers using boxes. Use your *OF* function inside a new function *HIST* to produce a horizontal histogram, as shown below:

```

      HIST 4 1 6 7
□□□□
□
□□□□□□
□□□□□□□

```

(*HINTS*: Use *OF* with a right argument of '□' on four different lines of the function to produce the display. Use indexing to pick numbers out of the right argument to *HIST* one at a time. The function should work for *four* numbers only; that is, *not* for three, five, and so on.)

- E. 1. Define a function *GT* that acts like this:

```

      SALES←125 350 201 115 279
      SALES GT 200
350 201 279

```

In other words, *GT* chooses numbers in the left argument greater than the number in the right argument.

---

Make sure that your function returns an explicit result, so the result can be stored in a variable:

```
        BIGSALES←SALES GT 200
        BIGSALES
350 201 279
```

2. Define an *LT* function similar to *GT*, except that it selects numbers less than the right argument. Then try the expression *SALES LT 200*.
3. Define an *EQ* function that selects numbers equal to the right argument. Then try the expression *SALES EQ 350*.

J F. Now it's time to create an actual system. (Don't panic! We'll give you lots of hints.)

The system should consist of four functions, called

- *INITBAL*
- *CHECK*
- *DEPOSIT*
- *NEWBAL*

These programs provide a system for

- setting an initial bank balance
- keeping a record of checks you write
- keeping a record of deposits you make
- showing the new balance after checks and deposits have been entered

All of the functions act on the same two variables, *DEPOS* (deposits) and *WITHD* (withdrawals).

We'll give you the first function:

```
        ▽INITBAL AMOUNT
[1]    DEPOS←AMOUNT
[2]    WITHD←0
        ▽
```

You would use the function one time only, to get started. (Or, you could use it at the end of each month to “clear out” your account and set the balance for the next month.) *INITBAL* sets *DEPOS* to the amount you say, and it sets *WITHD* to 0.

Here’s an example of using the system.

```
INITBAL 100
DEPOSIT 25 50
NEWBAL
175
CHECK 20 40 10
NEWBAL
105
```

*HINTS:*

1. *CHECK* and *DEPOSIT* should have the forms:

```
▽CHECK AMOUNT
▽DEPOSIT AMOUNT'
```

2. Remember that data can be added to the end of a vector using the catenate function. For example:

```
DATA←DATA,25
```

This expression catenates the number 25 to the end of an existing string of numbers.

3. *NEWBAL* should have the form:

```
▽NEWBAL
```

This function takes no arguments. It merely looks at *DEPOS* and *WITHD* and reports the present balance. (Use + / in this function.)

---

★ write function *DET* *M* which returns  
determinant of matrix *M*. (Hand-In)

---



## Chapter 6 Formatting

---

In this chapter, you'll learn how to use the format function and the system function `⎕FMT` to format data the way you'd like to see it. Using `⎕FMT`, you'll learn how to produce fancy reports and displays.

### WHAT IS FORMATTING?

We've seen that APL produces displays of our data in certain ways. This display of data is called *formatting*. You can control formatting to suit yourself using two tools: the format function (`⌘`) and the system function `⎕FMT`.

We've already explored one type of formatting with `format`; that is, we used it to change numeric data into character data. We can do even fancier things when we use `format` dyadically; that is, with a left and a right argument.

### DYADIC FORMAT

Just like monadic `format`, dyadic `format` turns numeric data in the right argument into character data. The difference is that the left argument tells `format` how to arrange the data.

### Format Pairs

The left argument consists of one or more pairs of numbers called *format pairs*. The first number controls the field width, that is, the number of columns that will contain the data. The second number specifies how to format the data. You can use one format pair to format all your data the same way, or you can use several pairs to format each column of data differently. Note: We're using the word *column* to mean a scalar, each element of a vector, or each column of a matrix.

Let's try formatting a vector.

```
DATA←1032 10.52 -20.11 3.1 2 -50
      6 0 ⌘ DATA
1032 11 -20 3 2 -50
```

### Formatting Whole Numbers

What happened? We used one format pair, `6 0`, to format each column, or element, in `DATA`. The first number, `6`, says "give each element six columns." But why did the decimal places disappear? They disappeared because the second number in the format pair control specifies the number of digits to the right of the decimal point.

In this case, the 0 says “round all the numbers with decimal places to the nearest whole number.”

### Formatting Numbers with Decimal Places

To keep decimal places in the data, substitute a positive number for the 0. The number you use corresponds to the number of decimal places you want.

```

      7 2 ¯ DATA
1032.00 10.52 -20.11 3.10 2.00 -50.00

```

You must remember to leave enough space for high minus symbols (in negative numbers), spaces (if you want them), and decimal points, or you’ll get a *DOMAIN ERROR*:

```

      6 2 ¯ DATA
DOMAIN ERROR
      6 2 ¯ DATA
      ^

```

### Formatting Columns Differently

What if we want to have different columns formatted differently? For example, suppose we want decimal places in some columns but not in others. In this case, you must specify a format pair for *each* column of data.

```

      6 0 6 2 7 2 4 1 2 0 4 0 ¯ DATA
1032 10.52 -20.11 3.1 2 -50

```

If you use too many or too few format pairs, you’ll get a *LENGTH ERROR*.

### Formatting a Matrix

You can also format a matrix the same way as a vector: specify one format pair for all the *columns* of the matrix or one pair for each column. Let’s turn *DATA* into a matrix:

```

      DATA←2 3 ρ DATA
      DATA
1032      10.52 -20.11
 3.1      2 -50

```

Negative → Efmt (scale?)  
no. sig. & 9.12

Formatting

Now, format it as whole numbers, using a field width of six:

| 6    | 0  | DATA |
|------|----|------|
| 1032 | 11 | -20  |
| 3    | 2  | -50  |

Ex:  $M \leftarrow 2 \text{ P } 3 \times 24$

$P \leftarrow \text{OUT} \leftarrow 8 \text{ P } M$

3.0E0 6.0E0  
9.0E0 1.2E1

Now, let's format each column differently:

| 6    | 0     | 6      | 2 | 7 | 2 | DATA |
|------|-------|--------|---|---|---|------|
| 1032 | 10.52 | -20.11 |   |   |   |      |
| 3    | 2.00  | -50.00 |   |   |   |      |

You can see that the first column has whole numbers with a field width of six, the second column has two decimal places with a field width of six, and the third column also has two decimal places, but with a field width of seven.

## FORMAT

Dyadic format gives us more control over the way we display our data. But, we may want more types of formatting than just controlling the placement of the data and the number of decimal places—especially for reports. In particular, we might want:

- commas
- dollar signs
- decimal points for dollar amounts

Accountants might want even fancier things, like parentheses around negative dollar amounts. All of this is easily done with the system function `FORMAT`.

note

`FORMAT` is not available on all APL systems. Refer to your system documentation or ask an experienced user if you don't get the answers you expect.

## FORMAT PHRASES

`FORMAT` takes "format phrases," surrounded by single quotes, as its left argument. Its right argument is data (scalars, vectors, or matrices). Common types of numeric formatting are:

`FM I W` integer (whole number)  
`FM F W S` floating point (numbers with decimal points)  
`FM E W S`  
`FM A W`  
`FM G <pattern>` a la COBOL

also T X  
text insertion

`r = rsc`  
`m = msc`  
`m < text`  
`m < text`  
`P < text`  
`P < text`

APL: (Quick) - P. 29

User Manual - ch. 8

You put a number on the right of the *I* or *F* to show how many columns you want. For example, *I6* means an integer in six columns. Let's try it with *DATA*:

```
'I6' □FMT DATA
1032      11      -20
   3       2      -50
```

Count the spaces, and you'll see that each number got six columns (the high minus counts as a column for negative numbers). Also, since we wanted integers, the numbers with decimal points were rounded.

```
'I10' □FMT DATA
1032      11      -20
   3       2      -50
```

This time, each number got 10 columns.

With *F*, you must indicate the total number of columns and also the number of columns to the right of the decimal point. (The decimal point counts as a column.)

```
'F8.2' □FMT DATA
1032.00    10.52  -20.11
   3.10     2.00  -50.00

'F8.3' □FMT DATA
1032.000   10.520 -20.110
   3.100    2.000 -50.000
```

If you don't give *□FMT* enough columns, you get stars where *□FMT* couldn't fit the numbers.

```
'F5.2' □FMT DATA
*****10.52*****
   3.10  2.00*****
```

## DECORATIONS

Decorations are also commonly used in formatting. The term "decorations" refers to items such as commas and dollar signs that are used to decorate data, making it easier to read and interpret. To get commas inserted in thousands, millions, and so on, use *C* in front of *I* or *F*:

`'CI10' □FMT 10000000`  
10,000,000

`'CF12.2' □FMT 10850.7`  
10,850.70

also **B** blanks zeros  
**Z** zero fill on left  
**L** left justify in field  
**R** = back ground fill

We'd also like decorations on either side of numbers, such as dollar signs, parentheses around negative numbers, and so on.

*M* <text>— Insert something on the left side of a negative number.

*N* <text>— Insert something on the right side of a negative number.

*P* <text>— Insert something on the left side of a positive number.

*Q* <text>— Insert something on the right side of a positive number.

Σ <is new>  
symbol pairs  
red. 9  
Z  
\* with

Let's try out these format phrases. We can use as many of them together as we like. Remember that each decoration counts as a column though, so we have to allow for enough columns or we'll get stars.

`'P<$>F8.2' □FMT 1082.2`  
\$1082.20

(The *text* in this case is a dollar sign.)

`'M<($>N<)>F10.2' □FMT -1082.2`  
(\$1082.20)

`'P<$>Q< >M<($>N<)>F15.2' □FMT 1002.6`  
345.32 -214.5  
\$1002.60  
\$345.32  
(\$214.50)

Σ with G - only  
replaces 9 + Z

As you can see from the last example above, vectors are "stacked" (displayed as a single column of numbers) when `□FMT` formats them.

\* However may use `□text□` with G format as

95 'G□99/99/99□' □FMT 21788

02/17/88

No % symbol in some APL  
[FASCO % key not ÷ or =]

Let's put percentage signs on the right of some numbers.

```
'N<•/•>Q<•/•>F8.1' □FMT 31.2 87.5 -23.8
31.2•/•
87.5•/•
-23.8•/•
```

To insert the letters *DR* and *CR* (for “debit” and “credit”) next to positive and negative numbers:

```
'M<>N< CR>Q< DR>F9.2' □FMT 32.34 -21.4
32.24 DR
21.40 CR
```

(The *M<>* part keeps the negative sign from printing.)

#### formatting and variables

Format strings can be assigned to variables, so that you can use them again and again. Let's assign some phrases to *FS* (for “format string”).

```
FS←'N<•/•>Q<•/•>F10.1'
FS □FMT 31.2 87.5 -23.8
31.2•/•
87.5•/•
-23.8•/•
```

#### formatting individual columns

If you join format phrases together with commas, you can format each column of a matrix differently.

```
DATA
1032 10.52 -20.11
3.1 2 -50

FS1←'M<($>N<)>P<$>F9.2,N<•/•>Q<•/•>
I7,M<>N<CR>F10.2'

FS1 □FMT DATA
$1032.00 11•/• 20.11 CR
$3.10 2•/• 50.00 CR
```

In the example above, we used three different format phrases, separated by commas, to format three columns of data. What if we only used two phrases?

Enter:

```
'I8,F8.2' □FMT DATA
1032    10.52    -20
 3      2.00    -50
```

Columns 1 and 2 are formatted using the phrases you entered. Then, □FMT goes back to the beginning of the format string and repeats the phrase to format column 3. If more data columns were included, the phrases would be repeated until all data columns were formatted.

### Repetition Factor

In the above example, we saw that format phrases will be repeated automatically, if we don't enter enough to handle all the data columns. A better way to repeat format phrases, however, is to use a "repetition factor." This factor allows you to control which phrases are repeated and how many times. The repetition factor is entered *before* the format phrase. Enter:

```
'I8,2F8.2' □FMT DATA
1032    10.52    -20.11
 3      2.00    -50.11
```

Since we included the 2 before the *F*, columns two and three were both formatted using the *F8.2* phrase.

### FORMATTING MULTIPLE PIECES OF DATA

□FMT can be used to format many different pieces of data at once. The data is enclosed in parentheses, and the pieces are separated from each other by semicolons (;). Enter:

```
NUMS←35.7 20.11

'2I8,2F8.2' □FMT (NUMS;DATA)
36    1032    10.52    -20.11
20      3      2.00    -50.00
```

The vector *NUMS* is stacked and formatted next to *DATA*.

formatting  
character  
data and  
numeric data  
together

Character data and numeric data can be formatted at the same time. To format a character matrix, use the format phrase *A* and a repetition factor equaling the number of columns in the character matrix.

```
MONTHS←2 3ρ'JANFEB'
      '3A1' ⍒FMT MONTHS
JAN
FEB

      '3A1,3F8.2' ⍒FMT (MONTHS;DATA)
JAN 1032.00    10.52    -20.11
FEB   3.10     2.00    -50.00
```

When you format character and numeric data together, it's important to get your repetition factors right, since the *A* format phrase is not defined for use with numbers.

Basic  
Reporting  
Functions

Let's use formatting inside a function to produce a report. Enter:

```
INCOMEDATA←3 3ρ10520.2 11500 12500
11217.71 11000 11500 0 0 0
INCOMEDATA
10520.2 11500    12500
11217.71 11000    11500
      0      0      0
      NAMES←3 7ρ'REVENUEEXPENSEPROFIT '
```

The first function, called *CALC*, calculates the profit by subtracting expenses (in row 2 of *INCOMEDATA*) from revenue (in row 1 of *INCOMEDATA*).

```
▽CALC
[1] INCOMEDATA[3;]←INCOMEDATA[1;]
   -INCOMEDATA[2;]
[2] ▽
```

The function simply subtracts the second row of *INCOMEDATA* from the first row. Then it assigns the answer to row 3 of *INCOMEDATA* (where we left 0s).



Now comes the actual report. Enter:

```

[1]      ▽REPORT
[2]      '                                HOMEGROWN CO.'
[3]      ' '
[4]      '                                1981                1982                1983'
[5]      ' '
[5]      CALC
[6]      '7A1,3CF12.2' □FMT (NAMES;INCOMEDATA)
[7]      ▽

```

*REPORT* prints some headings (centered), runs the *CALC* function, and then prints the body of the report. We get blank lines by putting a single blank ( ' ') on the appropriate lines of the *REPORT* function. Let's run the function:

```

REPORT                                HOMEGROWN CO.

                                1981                1982                1983

REVENUE    10,520.20    11,500.00    12,500.00
EXPENSE    11,217.71    11,000.00    11,500.00
PROFIT      -697.51      500.00      1,000.00

```

Of course, you could also add dollar signs, parentheses, and anything else you'd like.

## SUMMARY

In this chapter, you learned about the dyadic function *▽* and the system function *□FMT*. The general form of *□FMT* is

```

format string □FMT data
or
format string □FMT (data;data;...;data)

```

You learned how to use the following format phrases with `⎕FMT`.

| Format Phrase               | Description                                                                          |
|-----------------------------|--------------------------------------------------------------------------------------|
| <code>r Aw</code>           | Format character data                                                                |
| <code>C</code>              | Insert commas for thousands, millions, and so on                                     |
| <code>r Fw.d</code>         | Format floating-point data (numbers with decimal points)                             |
| <code>r Iw</code>           | Format integers (whole numbers)                                                      |
| <code>M &lt;text&gt;</code> | Place whatever is inside <code>&lt;&gt;</code> on the left side of negative numbers  |
| <code>N &lt;text&gt;</code> | Place whatever is inside <code>&lt;&gt;</code> on the right side of negative numbers |
| <code>P &lt;text&gt;</code> | Place whatever is inside <code>&lt;&gt;</code> on the left side of positive numbers  |
| <code>Q &lt;text&gt;</code> | Place whatever is inside <code>&lt;&gt;</code> on the right side of positive numbers |

Key: *r* means repetition factor; *w* means width (number of columns); *d* means number of decimal places.

`⎕FMT` offers many more capabilities in addition to those introduced here. Refer to your system documentation for more details. If you do not have an APL\*PLUS System, you may not be able to use `⎕FMT`, or it may work differently. Again, refer to your own system documentation for details.

EXERCISES

✓ A. Create the following variable:

```
D←2 3p10250.5 17512.7 18016.2
11205.81 16867.12 19197.2
```

Use □FMT to do the following exercises.

1. Format the numbers in *D* as integers, using a column width of eight.
2. Format *D* as floating-point numbers, using a column width of 10 and two decimal places.
3. Format *D* as floating-point, using a column width of eight and two decimal places.
4. Format *D* as floating-point, using a column width of 18 and two decimal places.
5. Repeat Exercise 4, using □FMT to insert commas in the data.
6. Repeat Exercise 5, using □FMT to place dollar signs next to the numbers.

B. Create the variables *E* and *N*:

```
E←2 2p18147.7 20.4 19717.11 -17.3
N←2 8p'REVENUESEXPENSES'
```

1. Format column 1 of *E* as floating-point, with a column width of 12 and two decimal places. Format column two of *E* as floating-point, with a column width of 10 and one decimal place.
2. Repeat Exercise 1, inserting commas and placing dollar signs next to the numbers in column 1.
3. Repeat Exercise 2, placing percentage signs (• / •) on the right of the numbers in column 2.
4. Repeat Exercise 3, changing the APL high minus (the one next to the negative percentage) to the more familiar hyphen.

5. Repeat Exercise 4, placing a plus sign (+) next to the positive percentage.
6. Repeat Exercise 5, formatting  $N$  next to  $E$ . That is, format  $N$  first, then  $E$ , using *one*  $\square FMT$  statement. The result should look like this:

```
REVENUES    $18,147.70    +20.4% / °
EXPENSES    $19,717.11    -17.3% / °
```

- C. Create the variables  $S$  and  $N1$ :

```
S←2 2p6543.1 5342.55 6231.89 7531.21
N1←2 9p'SALESREP1SALESREP2'
```

1. Format the numbers in  $S$  as floating-point, with a column width of 12 and two decimal places.
2.  $S$  contains the sales figures for two sales representatives for two weeks. The rows correspond to the sales representatives; the columns to the weeks. We can find the total sales for two weeks like so:

```
      T←+ / S
      T
11885.65 13763.1
```

Repeat Exercise 1, formatting the totals ( $T$ ) next to  $S$  (on the right-hand side of  $S$ ). Use the same format phrase for both pieces of data.

3. Repeat Exercise 2, inserting dollar signs next to all the numbers.
4. Repeat Exercise 3, inserting commas in the thousands position.
5. Format  $N1$  on the left of the numbers. (*HINT*: Remember to use repetition factors for *both* format phrases.)

- D. 1. Define a function *REPT* that produces a report using the information from Exercise C.5. The function should have the syntax *REPT data*. *REPT* finds row totals for *data* using  $+/$ , stores the totals in a variable called *TOT*, then formats *N1*, *data*, and *TOT* together. Test your function on *S* by entering:

*REPT S*

2. Define a new function *RPT2* that behaves the same as *REPT*, except that it prints the centered title *SALES FIGURES* and the column titles *WEEK1*, *WEEK2*, and *TOTALS* before it formats the body of the report.

Hand-In

Show function  
and output

E. Generate Table of <sup>integer</sup> ~~Squares~~ <sup>Squares + SS roots</sup> ~~+ roots~~, in Scaled form with 5 sig. digits, for integers ~~from~~ <sup>last</sup>  $A=100$  thru  $B=130$ , incl. vis.

| X    | X SQ     | SQRT X   |
|------|----------|----------|
| 100  | 1.0000E4 | 1.0000E1 |
| etc. |          |          |

Note ~~DPGE (CP-6) skip to Next Page~~

~~DPFC 'N' (N = right format ctrl per CP-6 Prog Ref Manual)~~

~~Next Print Line only~~

A = print, then adv 1  
B = " " " 2  
f = adv 1 then print  
b = adv 2 " " } may cause problems on video term

APL Plus Ver 10  
Quick Ref

p. 94;

~~DTLFF (limited)~~

~~DT~~ (3 elem.)  
DT = TCBS  
DT = TC NL (new line = CR)  
DT = TCLF

103  
DTESC = ESC = [AV[DT0+27]]

## Chapter 7

# Storage Facilities: Workspaces

---

In this chapter, you will learn what a workspace is and how functions and variables can be “permanently” stored in a workspace. You’ll also learn how to list the names of your workspaces, change the name of a workspace, erase individual functions and variables in the workspace, and erase entire workspaces.

### WHAT IS A WORKSPACE?

A workspace is a place in the computer where work is done and objects (functions and variables) can be stored. You’ve been working in workspaces all this time—either in “clear” ones or in ones you’ve named. You can think of workspaces as different drawers of a desk you work at. As you need to do different work, you get things out of different drawers. You have to replace the things in the drawers when you’re finished or they get lost.

### note

On most APL systems, a collection of workspaces is called a “library,” and these libraries are differentiated by library numbers. On most mainframe APL systems, the libraries are logical collections of related workspaces. On some personal computer APL systems, the library numbers refer to disk drives, either physical or logical. Some personal computer APL systems use disk names instead of library numbers. If you omit the library number when you type the name of a workspace, APL assumes you want a workspace in the current library or on the current disk. Refer to your system documentation or ask an experienced user to find out how your APL system handles libraries or disks.

### active workspace

In APL, we “get things” using the command `LOAD` and “put them back” using the command `SAVE`. We can move back and forth between workspaces by using these commands together. We load one workspace, save it when we’re finished, then load another, save it, and so on.

Whenever you’re in APL, you’re *always* using a workspace. We call the workspace you’re in your “active workspace” (as opposed to stored workspaces). Your active workspace may be a clear one that has no name, or it may already have a name. Most systems give you a clear workspace when you sign on. You can always give it a name by storing it under a name you choose. For example, enter:

```
    )SAVE MYWORK  
SAVED . . .
```

When you enter this, a message is printed telling you what the workspace name is and that it's been stored. (We won't show the entire message, since it's slightly different on each system. On some systems, the message includes only a date and time and no *SAVED* part.)

No matter which workspace you're in, all of the built-in APL functions (+,  $\div$ ,  $\times$ ,  $\div$ ,  $\rho$ , and so on) are always available for you to use.

## SYSTEM COMMANDS

Besides saving and loading workspaces, we can do other things with workspaces, such as listing the things contained in them, erasing things, and copying things from other workspaces. The commands that allow us to do these things are called "system commands." We use this term because these commands control the system rather than doing calculation. All of them begin with a right parenthesis.

### )SAVE

The system command *)SAVE* stores a copy of the workspace you're in under a name you give. It takes the form:

```
    )SAVE name
```

Enter:

```
    )SAVE EXAMPLE  
SAVED . . .
```

A message is displayed telling you when the workspace was stored, and so on. You cannot save a clear workspace; you'll get an error message. Be sure to name your clear workspace when you save it.

Notice that a *copy* of the workspace is stored. When you enter *)SAVE* the active workspace remains the same.

If the workspace already has a name, you only need to enter:

**)SAVE**

The workspace will be saved under its “old” name. Enter:

**)SAVE**  
*SAVED . . .*

The *SAVED* message includes the name of the workspace, which is the name you gave it a moment ago—*EXAMPLE*.

The most important fact you should remember about **)SAVE** is that you can create variables and functions in your active workspace, but *none* of these are permanently stored until you use the **)SAVE** command.

**)LOAD** is used to get into a workspace you’ve stored already. It takes the form:

**)LOAD    *name***

Let’s load a workspace:

**)LOAD DEMOAPL**  
*SAVED . . .*

If the *DEMOAPL* workspace is not in the current library or on the current disk, you will have to type in the appropriate library number or disk name, depending on your APL system. You can now run programs in this workspace the way you did in Chapter 1.

Again, the thing to remember is that, although you’ve loaded the workspace, none of the things you create in it will be stored until you use the **)SAVE** command.

**)CLEAR**

**)CLEAR** puts you in an empty, unused workspace. You’ll want to do this if you want to do some new work and keep it separate from other work. Just create your new functions and variables; then save the workspace under a new name.



To see how this works, let's define a few variables and a function:

```
A ← 5
B ← 19
▽X ADD Y
[1] X + Y
[2] ▽
```

Let's save the workspace under the name *EXAMPLE*:

```
    )SAVE EXAMPLE
SAVED . . .
```

Now let's clear the workspace:

```
    )CLEAR
CLEAR WS
```

We see that *A*, *B*, and *ADD* are no longer there:

```
      A
VALUE ERROR
      A
      ^
      7 ADD 8
SYNTAX ERROR
      7 ADD 8
      ^
```

)FNS

)FNS gives an alphabetical listing of the names of the functions in the workspace.

)VARS

)VARS provides an alphabetical list of the names of the variables in the workspace. You've used )FNS and )VARS before, but let's get back into the workspace *EXAMPLE* to use them there.

```
    )LOAD EXAMPLE
SAVED . . .
    )FNS
ADD
    )VARS
A      B
```

**)ERASE**

*not erase total VARS  
inside func*

**)ERASE** is used to delete variables and functions from the workspace. It takes the form

**)ERASE** *object1 object2 . . .*

Let's try it:

**)ERASE A  
      )VARS**

**B**

Again, the deletions you make are in the active workspace and don't affect the stored copy of the workspace unless you do a **)SAVE**. We can prove this to ourselves by loading *EXAMPLE* again:

**)LOAD EXAMPLE  
SAVED . . .  
      )VARS  
A       B**

To erase something in the workspace "forever," you need to use **)ERASE** and then **)SAVE** right afterwards.

*not*  
**)WSLIB or  
      )LIB**

**)WSLIB** or **)LIB** (depending on your APL system) lists the workspaces you have in your library or on your disk. Let's try this command:

**)WSLIB**

(We won't show the list, since it will be different depending on the workspaces you've stored. The list should include *EXAMPLE* at a minimum though.)

**)WSID**

**)WSID** gives the name of the active workspace (the one you're presently working in). Let's try it:

**)WSID  
IS EXAMPLE**

Let's see what happens when you clear the workspace:

```
        )CLEAR
CLEAR WS
        )WSID
IS CLEAR WS
```

You can also use the `)WSID` command to rename your active workspace:

```
        )WSID EX1
WAS CLEAR WS
        )WSID
IS EX1
```

Then later you can just enter `)SAVE` to save the workspace, since it's already got a name.

`)DROP`

`)DROP` deletes one of your stored workspaces. It has the form:

```
        )DROP name
```

The named workspace is dropped from storage. This command does *not* affect the active workspace. Let's define something in `EX1`.

```
XX←111
```

Then let's save `EX1`:

```
        )SAVE
SAVED . . .
```

If you enter `)LIB` or `)WSLIB` now, you'll see that `EX1` has been added to your list of stored workspaces. Now let's drop it:

```
        )DROP EX1
```

If you enter `)WSLIB` now, you'll notice the `EX1` has been deleted from the library. But notice that you're still inside `EX1`:

```
        )WSID
IS EX1
```

---

Also notice that *XX* is still around:

```
      XX
1 2 3 4 5 6 7 8 9 10 11
```

If you load another workspace or clear the workspace though, *EX1* will be gone forever, since it's no longer stored.

```
      )CLEAR
CLEAR WS
      )LOAD EX1
WS NOT FOUND
```

**)COPY** *)COPY* is used to copy objects (functions and variables) from one workspace into another. It takes the form:

```
      )COPY wsname objectnames
```

For example, enter:

```
      )COPY MYWORK SALES
SAVED .
      SALES
125 350 201 115 279
```

Your values for *SALES* may be different if you changed *SALES* in this workspace since Chapter 5. If you never stored *MYWORK*, you'll get the message *WS NOT FOUND*. You can easily copy something stored in one of the workspaces listed by *)WSLIB* though.

If you omit the *objectnames* in the *)COPY* command, the entire workspace is copied into your active workspace. Enter:

```
      )COPY MYWORK
SAVED . . .
```

If you list the variables now, you'll see the entire list from *MYWORK*.

Again, remember that the `⋄COPY` command only brings things to your active workspace. Only the `⋄SAVE` command can store the new items in the workspace in your library.

**⋄OFF**

`⋄OFF` signs you off from the APL system. (On personal computer systems, it's important that you use this command if you have any files active—which we'll learn about in Chapter 8—because this protects the files.)

**SYSTEM  
FUNCTIONS**

Part of the nature of system commands is that they can't be used inside functions. This probably doesn't concern you too much right now. But when you've gained some experience with APL, you'll want to use commands like `⋄LOAD` under program control. You can do this with "system functions," all of which begin with a quad (`⍷`). For example, `⍷LOAD` does the same thing as `⋄LOAD`. Some system functions have slightly different names than the corresponding system commands. For example, `⍷EX` does the job of `⋄ERASE`. Also, there are many system functions that do entirely new things that no system command can do. One of these is `⍷FMT`, which you learned about in Chapter 6.

Surprise! There are no exercises for this chapter. You should use the commands you've learned to play around with your stored workspaces, but take it easy with `⋄DROP`.

**SUMMARY**

Below is an alphabetical list of the system commands you learned in this chapter and a short explanation of each command.

|                                   |                                                                             |
|-----------------------------------|-----------------------------------------------------------------------------|
| <code>⋄CLEAR</code>               | Clears the active workspace—provides you with an empty active workspace     |
| <code>⋄COPY wsname</code>         | Copies the contents of an entire workspace into the active workspace        |
| <code>⋄COPY wsname objects</code> | Copies the named objects from the named workspace into the active workspace |
| <code>⋄DROP wsname</code>         | Drops (deletes) a workspace from storage                                    |
| <code>⋄ERASE objects</code>       | Erases the named objects from the active workspace                          |

---

|                             |                                                                        |
|-----------------------------|------------------------------------------------------------------------|
| <code>)FNS</code>           | Lists the names of functions in the active workspace                   |
| <code>)LOAD wsname</code>   | Clears the active workspace and loads a copy of the stored workspace   |
| <code>)OFF</code>           | Signs you off of the APL system                                        |
| <code>)SAVE</code>          | Saves (stores) a copy of the active workspace under its present name   |
| <code>)SAVE wsname</code>   | Saves (stores) a copy of the active workspace under the name indicated |
| <code>)VARS</code>          | Lists the names of variables in the active workspace                   |
| <code>)WSID</code>          | Shows the name of the active workspace                                 |
| <code>)WSID wsname</code>   | Changes the name of the active workspace to the name indicated         |
| <code>)WSLIB or )LIB</code> | Lists the workspaces in your library or disk                           |

## Chapter 8

### Storage Facilities: Files

---

In this chapter, you will learn what a file is. You will also learn how to create files, store data in them, retrieve data from them, and erase them when you no longer need them.

#### note

Some APL systems do not have a file facility like that described in this chapter. Refer to your system documentation or ask an experienced user if you don't get the answers you expect.

#### WHAT IS A FILE?

Files provide data storage outside of workspaces. A major difference between files and workspaces is that only *data* can be stored in files, while workspaces can store both functions and data. This means that no actual work (computation) goes on inside files. Instead, you get the data from a file (that is, bring a copy of it into your active workspace), work on it, and then put it back in the file, if necessary.

If the active workspace can be thought of as a desk, a file can be thought of as a filing cabinet in the same room. If you want something from the desk, you have it right at hand. But if you want to work on something in the filing cabinet, you have to get up and get it. And if you want to make sure the stuff doesn't get lost, you have to go put it back in the filing cabinet when you're finished.

You may not have any need for a filing cabinet if you don't have many things to store. Your desk has a good amount of room for storing things. But if you need to store a lot of stuff, you're better off putting it in the filing cabinet.

It's the same in APL. If you don't have much to store, you can store it in your workspace. But if you have lots of data, it will be more convenient to put it in a file. Files are also nice for organizing your data.

#### ADVANTAGES OF USING FILES

Files have many advantages over workspaces. The main ones are:

- Files can hold much more data than a workspace.
- Files remain active when you change workspaces.
- Files give you extra storage without "stealing" any room from your workspace.

## CHARACTER- ISTICS OF FILES

Files have different characteristics than workspaces. The characteristics of files are listed below.

### components

- You don't automatically get files as you do an active workspace—you must create files.
- A file is composed of *components*. Each component can contain something different, as does a variable.
- Components can contain numeric or character data of any size or shape—from single numbers to matrices. Some components may even be empty.
- A component contains one thing at a time.
- When a file is first created, it has no components and is said to be empty.
- You add components to the file by *appending* them to the end of the file.

### component numbers

- Rather than having a name as variables do, a file component is identified by its number. The system numbers the components sequentially; that is, 1, 2, 3, 4, and so on. This makes it convenient to refer to the components by their numbers.
- We change the data in file components by *replacing* them with new data. The data can be slightly different than the old data (catenate one element to the end of a vector and replace the old vector with it) or completely different (replace a single number with a matrix or with character data).

### tie numbers

- To be active, a file must be associated with a number. We say that the file is *tied* to the number. This makes it convenient to refer to the entire file by its number. For instance, the expression to read the first component of a file tied to the number 33 is

`⊞FREAD 33 1`

## FILE FUNCTIONS

All functions that deal with files begin with the characters `⊞F`. The `⊞` means that they are system functions (which we mentioned in Chapter 7), and the `F` stands for “file.” There are almost two dozen functions to use with files, but we will look only at the simplest and most useful ones.



**□FCREATE**

You can create a file using the command **□FCREATE**.  
It has the format:

**'name' □FCREATE tie number**

The *name* is surrounded by single quotes because it is character data. The name can contain both letters and numbers, but it must begin with a letter. Different APL systems allow different lengths for file names; refer to your system documentation. The *tie number* is any positive whole number you choose.

Let's create a file. Enter:

**'TEST' □FCREATE 1**

We know that the file has been created because no error message appears and the system returns us to the six-space indent.

If we try to create another file with the same name, we get an error message.

```
      'TEST' □FCREATE 2
FILE NAME ERROR
      'TEST' □FCREATE 2
      ^
```

Thus, you can see that it's important for each file to have a unique name.

**□FAPPEND**

Let's put some data in the file. Any old data will do. Let's create a variable:

```
      VAR←15
      VAR
1 2 3 4 5
```

Now let's append it to the file:

```
      VAR □FAPPEND 1
1
```

We know this was successful since the computer displayed the number of the component it added to the file (1). (We'll omit this number in the following examples.)

`⊞FAPPEND` has the format:

*data* `⊞FAPPEND` *tie number*

The *data* can be either some numbers or some character data, or the name of a variable that contains numbers or characters. If *data* is character data, it must be enclosed in single quotes.

Let's append some more data to our *TEST* file:

```
22 ⊞FAPPEND 1
'THIS IS A TEST.' ⊞FAPPEND 1
DATA←3 3ρ'ONETWO3 '
DATA
ONE
TWO
3
DATA ⊞FAPPEND 1
```

`⊞FSIZE`

Now let's check the file's size. Enter:

```
⊞FSIZE 1
1 5 512 0
```

This function has a simple format:

`⊞FSIZE` *tie number*

Its result shows:

1. The starting component number of the file (usually 1).
2. The number of the component that will be filled (appended) *next*.
3. The space (in bytes) taken up by the file.
4. The growth limit for the file. A 0 means "no limit"—the file can grow to virtually *any* length (that is, until the computer you are working with runs out of storage).

Items 3 and 4 will vary depending on the system you're using. Note that item 2 shows the *next* component to be filled. Since the size of our file is 15, we know that components 1 through 4 have already been used, and component 5 will be the next component we append.

`□FREAD`

Now let's read the stuff we've stored. Enter:

```
      □FREAD 1 3
THIS IS A TEST.
```

This result can be stored in a variable for further work:

```
      S←□FREAD 1 3
      S
THIS IS A TEST.
(S≠' ')/S
THISISATEST.
```

We can change what we read:

```
      S←4ρS
      S
THIS
```

But this doesn't affect the things in the file:

```
      □FREAD 1 3
THIS IS A TEST.
```

`□FREAD` has the following general form:

`□FREAD    file number component number`

If you try to read a component that doesn't exist, you get an error message:

```
      □FREAD 1 72
FILE INDEX ERROR
      □FREAD 1 72
      ^
```

## `⎕FREPLACE`

To change a component in a file, you must *replace* the component:

```
'TEST NUMBER TWO' ⎕FREPLACE 1 3
```

This function has the format:

```
data ⎕FREPLACE tie number component number
```

If we read component 3 now:

```
⎕FREAD 1 3  
TEST NUMBER TWO
```

we see it's been changed.

## `⎕FUNTIE`

To deactivate a file, use `⎕FUNTIE`, which “unties” the file.

```
⎕FUNTIE 1
```

We can't read the file now:

```
⎕FREAD 1 3  
FILE TIE ERROR  
⎕FREAD 1 3  
^
```

This function has the format:

```
⎕FUNTIE tie number(s)
```

It's important to untie all files *before* removing a disk or resetting the system. On all systems, the `⌵OFF` command automatically unties any files currently tied.

**␣FTIE**

To activate a file that's already been created, you must tie it:

```
'TEST' ␣FTIE 7
```

We used a different tie number this time. Any whole number will do, as long as it's not already being used. (If you're already using the number for another file, you'll get a *FILE TIE ERROR*.) We can read data now:

```
␣FREAD 7 3  
TEST NUMBER TWO
```

Again, tie numbers are only *temporary* identifiers for a particular file. You can use a different tie number each time you tie the file. But if you tie many files, you must use a different tie number for each one. This is because the system wouldn't know which file you mean when you enter `␣FREAD 7 3` if two different files were tied to the number 7.

**␣FNAMES**

To find out what files you have tied, enter:

```
␣FNAMES  
TEST
```

A library number or disk name might be printed before the file name, depending on your APL system.

**␣FNUMS**

To find out what numbers your files are tied to, enter:

```
␣FNUMS  
7
```

If you have more than one file tied, `␣FNUMS` will return a vector of tie numbers. The order of the vector corresponds to the order of the list of file names produced by `␣FNAMES`.

A good technique for untying *all* files is

```
␣FUNTIE ␣FNUMS
```

since `␣FNUMS` contains the tie numbers of all your files.

## **⎕FERASE**

To erase a file you no longer need, use **⎕FERASE**. Enter:

```
'TEST' ⎕FERASE 7
```

This function has the format:

```
name ⎕FERASE tie number
```

The file must be tied in order for you to erase it. Also, you must give both the file name and its tie number (to make sure you don't use the wrong tie number and erase the wrong file). The library number or disk name is required before the file name.

**⎕FNAMES** now shows that nothing's tied:

```
⎕FNAMES
```

Let's create one more file. Enter:

```
'TEST' ⎕FCREATE 22
```

This new file can have the same name as the old file, since the old one no longer exists. You can untie this file now:

```
⎕FUNTIE 22
```

## **⎕FLIB**

**⎕FLIB** shows what files you have. It has the form:

```
⎕FLIB identifier
```

The argument *identifier* is the library number or disk name. For example, on the APL\*PLUS System for the PC, to list the files on the disk in the B disk drive, you would enter:

```
⎕FLIB 1  
1 TEST
```

Other file names could also display, depending on which disk was in the disk drive.

On some APL systems, the system command `)FLIB` will list your files.

`)FLIB`

1 `TEST`

## SUMMARY

In this chapter, you learned how to use the file system functions shown in the following table. Note that many of these commands presume that you have a file tied. If you get a `FILE TIE ERROR`, it's probably because you don't have a file tied.

| File Function                                                                                                                                            | Description                                  |
|----------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------|
| <code>data</code> <code>⊞FAPPEND tie number</code>                                                                                                       | Append data to the end of a file             |
| <code>name</code> <code>⊞FCREATE tie number</code> <code>⊞FOPEN</code> <sup>CREATE</sup> <sub>(new file)</sub>                                           | Create (and tie) a file                      |
| <code>name</code> <code>⊞FERASE tie number</code>                                                                                                        | Erase a file                                 |
| <code>⊞FLIB ''</code> or <code>⊞FLIB identifier</code>                                                                                                   | List the files in a library                  |
| <code>⊞FNAMES</code> ✓                                                                                                                                   | List the names of files currently tied       |
| <code>⊞FNUMS</code> ✓ <sub>(= ⊞FNUMS)</sub>                                                                                                              | List the tie numbers of files currently tied |
| <code>⊞FREAD tie number component number</code>                                                                                                          | Read the data in one component of a file     |
| <code>data</code> <code>⊞FREPLACE tie number component number</code>                                                                                     | Replace data in a component with new data    |
| <code>⊞FSIZE tie number</code>                                                                                                                           | Show the size of a file                      |
| <code>name</code> <code>⊞FTIE tie number</code> <sup>⊞FOPEN</sup> <sub>⊞FTIE</sub> <sup>⊞FOPEN</sup> <sub>⊞FTIE</sub> <sup>⊞FOPEN</sup> <sub>⊞FTIE</sub> | Tie a file                                   |
| <code>⊞FUNTIE tie number(s)</code> <code>⊞FCLOSE</code> <sub>⊞FUNTIE</sub>                                                                               | Untie a file or files                        |

## EXERCISES

- A. 1. Create a file named *EXERCISE*.
2. Create a variable called *MSG* containing the message:

*FILE FOR EXERCISES IN CHAPTER 8.*

This is just an informative message to identify the file. Append it to the file.

3. Create a variable called *DATA*—a 3 by 3 matrix containing the numbers 110, 150, 117, 35, 65, 79, 84, 114, 97. Append it to the end of the file.
4. Create a variable called *DATA2*—a vector containing the numbers 11.52, 3.72, 6.15. Append it to the end of the file.

5. Read in the data in component 2 and assign it to a variable named *ITEMS*. (Remember, this can be done with a single APL statement.)
  6. Read in the data from component 3 and assign it to a variable named *PRICES*.
  7. Write an APL expression to find the third price in *PRICES*. (*HINT*: Use indexing.)
  8. Write an APL expression to find the third row of *ITEMS*.
  9. We sell nuts, bolts, and wrenches in our store. Row 1 of *ITEMS* shows how many nuts we sold in each of the past three weeks, and rows 2 and 3 show the same thing for bolts and wrenches. Write an APL expression to find the *total* number of nuts, bolts, and wrenches we sold in the past three weeks. (*HINT*: Use plus reduction.)
  10. Write an APL expression to find the total number of *wrenches only* sold during the three weeks. That is, find the row total for row 3 of *ITEMS* only.
  11. *PRICES* contains the prices we charge for our nuts, bolts, and wrenches. Write an APL expression to find out how much we charged for all the wrenches we sold during the past three weeks.
- B.
1. Untie your *EXERCISE* file.
  2. Define a function called *TIE* that will tie the *EXERCISE* file to the number 277.
  3. Execute the function *TIE*. Check to see whether it worked by entering  $\square FNAMEs$  and then  $\square FNUMs$ .
  4. Define a function called *READ* that displays the number of sales for the past three weeks from the file *EXERCISE*. That is, it displays the entire matrix stored in component 2. (Remember that the file is tied to 277.) Run the function.
-



5. Write a function called *TOTSALES* that reads in both components 2 and 3. It then totals the unit sales given in component 2 (finds total sales for each item for three weeks). Finally, it multiplies the totals by the prices to find the total sales. Run the function. (*HINTS*: In your function, assign the unit sales to the variable *ITEMS*, and the prices to the variable *PRICES*. The rest of the function is similar to Exercise A.11.)
- C. File system functions work with variables much as they work with numbers. For example, you can enter:

```
TIENO←1
'TEST' □FTIE TIENO
□FNUMS
```

1

(If you didn't untie the *TEST* file before entering this, you may have gotten an error message.)

```
1000 □FAPPEND TIENO
□FREAD TIENO,1
1000
```

(The comma combines the variable *TIENO* with the 1.)

Since file functions can use variables like this, you can assign component numbers to meaningful names and then use the names to read data. Enter:

```
M←'FILE FOR TESTING EXAMPLES.'
M □FREPLACE TIENO,1
MESSAGE←1
□FREAD TIENO, MESSAGE
FILE FOR TESTING EXAMPLES.
□FREAD 1 1
FILE FOR TESTING EXAMPLES.
```

You can put all of this in a function to make things really “friendly.”  
First, let’s erase *READ* :

**)ERASE READ**

Then define a new *READ* function:

```
      ∇READ DATA
[1]   □FREAD TIENO, DATA
[2]   ∇
```

Now enter:

**READ MESSAGE**  
**FILE FOR TESTING EXAMPLES.**

1. Enter **)CLEAR** to clear your workspace. Define a *READ* function to read in data from your exercise file. You should be able to do the following:

```
      READ ITEMS
110 150 117
 35  65  79
 84 114  97
      READ PRICES
11.52 3.72 6.15
```

[*HINTS*: This function will be similar to the *READ* function we defined above, but it will use the tie number for *EXERCISE* (277). Remember to assign the correct component numbers to the variables *PRICES* and *ITEMS* before trying to run *READ*.]

2. Write a function *TOTAL* that reads in a matrix and totals the rows in the matrix. You should be able to read in your unit sales data from *EXERCISE* and total it like so:

```
      TOTAL ITEMS
377 179 295
```

3. Write a function called *TOTSALE* that shows how much we made on a particular item in each of the three weeks. That is, the result should be three numbers—each one is the unit sales for the week times the price of the item. You should be able to get the following results with your function:

```

      TOTSALE NUTS
1267.2 1728 1347.84
      TOTSALE BOLTS
130.2 241.8 293.88

```

The function should:

- read in the unit sales data
- read in the prices
- find the correct row in the unit sales matrix *ITEMS*
- find the correct price in the variable *PRICES*
- multiply the price by the correct row

(HINTS: 1. To find the correct row, use a variable together with indexing:

```

      D←3 5p115
      D
  1  2  3  4  5
  6  7  8  9 10
11 12 13 14 15
      X←2

```

```

      D[X;]

```

```

6 7 8 9 10

```

This also works for vectors:

```

      E←15
      E[X]
2

```

2. To make your function work, you have to assign the row numbers of items to variables:

```
NUTS←1
BOLTS←2
WRENCHES←3 )
```

- D. 1. Create a file called *SALES*.

2. Create the following variables.

```
SNOS←101 102 103
MON←1 2 3 4
SALES←3 4 107 62 97 108 80 72
95 67 88 92 101 81
```

Append these variables to the *SALES* file (first *SNOS*, then *MON*, then *SALES*).

*SNOS* are salesman numbers, and *MON* are the numbers for four months. *SNOS* corresponds to the rows of *SALES*; *MON* corresponds to the columns.

3. Untie the *SALES* file.
4. Write a function called *GETDATA* that:
- Ties *SALES* to the number 1076.
  - Reads in each of the three components in *SALES* and assigns them to the variables *SNOS*, *MON*, and *SALES*. (This is done so that the data will be in the workspace where we can work on it—there's no need for *every* function to read components from the file.)

Run the function.

5. Enter the following function *SALESMAN*, which assigns the number you give it to a variable *S*.

```
      ▽ SALESMAN NUM
[1]   S ← NUM
[2]   ▽
```

6. Write a function *MONTH* that is similar to the *SALESMAN* function. *MONTH* assigns the number you give it to a variable *M*.
7. Write a function *PRINT* that uses *M* and *S* to find the desired element of *SALES*. The functions should behave like this:

```
      SALESMAN 101
      MONTH 4
      PRINT
108
```

The function *PRINT* chose the row for salesman 101 (row 1) and the column for month 4 (column 4).

[*HINT*: Use the selection techniques you learned in Chapter 4. For example, to get the result shown above, you could use

```
      (MON=4) / (SNOS=101) / [1] SALES
108
```

Remember that 101 and 4 were stored in variables by the functions *SALESMAN* and *MONTH*, though.

Notice that you've created a complete system. A complete (if short!) run using this system might go like this:

```
      GETDATA  
      SALESMAN 103  
      MONTH 1  
      PRINT  
88  
      )OFF  
FILES UNTIED
```

If someone created this system for you, you could use it without ever knowing what a file is.]

## Chapter 9

# APL Programming Techniques

---

In this chapter, you will learn how to define “talkative” functions; that is, functions that will display messages requesting the information they need to complete a task. You will also learn about some more built-in APL functions—take, branching, and catenation with matrices. You will be introduced to some APL programming aids and see a sample application that stores and retrieves recipes in a file.

### INTERACTIVE FUNCTIONS

Up to this point, we’ve used functions that get their data when we enter their names. For example,

```
      AVG 8 10 12
10
      4 PERIMETER 6
20
```

Some programs need to ask us for the data they need and then wait for us to enter it. For example,

```
      ASKAVERAGE
WHAT NUMBERS DO YOU WANT TO AVERAGE?
□:
      8 10 12
THE AVERAGE IS 10.
```

We call functions that ask for their data *interactive*, since they interact with you.

### Evaluated Input

Getting *ASKAVERAGE* to display a question is easy—we already know how to use character data in a function. The new part is the `□ :`. We say that the computer is in “evaluated input mode” when it prints this symbol. (This is known informally as “quad input.”) It is waiting for us to enter numbers (or a variable containing numbers) that it will use for data. We cause the computer to do this by including the statement:

```
variable ←□
```

in our function. Of course, *variable* can be any variable name you choose. Let's create the function *ASKAVERAGE*:

```
      ▼ ASKAVERAGE
[1]   'WHAT NUMBERS DO YOU WANT TO AVERAGE?'
[2]   VECTOR←□
[3]   ANSWER←(+/VECTOR)+ρVECTOR
[4]   'THE AVERAGE IS ',(▼ANSWER),'. '
[5]   ▼
```

Notice that the function needs no arguments, since it gets its data by asking for it. Notice also that we've left a blank between the del (▼) and the function name. This is perfectly okay, and it's easier to read, so this is the style we'll use from now on.

You can now go back and rewrite any of the functions we've created so far using evaluated input.

## Character Input

Quad input works fine for numbers, but what about characters? Quad input will work with character data, but only if the person using the function remembers to enclose the data in single quotes. Since most casual users won't know that character data should be enclosed in quotes, you'll want to use a programming technique that will allow them to enter character data without the quotes. This technique is called "character input mode" or "quote-quad input mode." You'll see in a minute where the "quote-quad" terminology comes from.

Quote-quad input accepts letters and other characters without quotes. For example,

```
      HELLO
MY NAME IS JERRY.  WHAT'S YOURS?
JIM
NICE TO MEET YOU, JIM.
```

Notice that quote-quad input mode doesn't cause the computer to display any special symbol. The computer merely places you on the next line (*without* a six-space indent) and waits for you to enter something.



Let's create the function *HELLO*:

```
▽ HELLO
[1]  'MY NAME IS JERRY.  WHAT''S YOURS?'
[2]  A←⍴
[3]  'NICE TO MEET YOU, ',A,'.'
[4]  ▽
```

Looking at the function *HELLO*, you can now see where the term “quote-quad” comes from. The symbol that you use in your function to accept character data without quotes is a quote-quad (⍴), which looks like a quad with a quote inside it. Using the quad and quote-quad input techniques, you can make your functions much friendlier and more flexible.

#### note

In most cases you can interrupt your function to make a function stop (if you want to get out of it for any reason). It turns out that this *doesn't* work with evaluated input or character input. The system is so set on your entering something that it won't let you get away with merely interrupting input. It keeps printing ⍴: , or jumping to the next line, until you actually enter something.

To escape from evaluated input mode, simply enter a right arrow (→). The techniques for escaping from character input mode vary among APL systems. Refer to your system documentation or ask an experienced user. When you do escape from either mode, you return to immediate execution mode.

The next sections introduce some additional built-in APL functions that will help you make full use of the quad and quote-quad techniques.

#### CATENATION WITH MATRICES

One APL function useful with character data is catenation. We learned in Chapter 3 that it can be used with vectors:

```
V1←'HAM'
V2←'BURGER'
V1,V2
HAMBURGER
```

However, matrices have two dimensions, so we have to tell APL which dimension we want to connect the matrices along. (This is exactly like what you learned to do with reduction and compression.)

```
      M1←3 3ρ'HISHERMY '
      M2←3 3ρ'CARBARTAR '
      M1,[1]M2

HIS
HER
MY
CAR
BAR
TAR
```

As in the case of reduction and compression, catenate with matrices selects the *last* dimension (rows) if you don't give it a specific dimension.

```
      M1,M2

HISCAR
HERBAR
MY TAR
      M1,[2]M2

HISCAR
HERBAR
MY TAR
```

That is,  $M1,[2]M2$  is the same as  $M1,M2$  in this case.

## TAKE

Take does exactly what its name says—it “takes” as many elements from its vector right argument as you tell it in the left argument.

```
      VECT←1 10
      4↑VECT
1 2 3 4
      S←'ONCE UPON A TIME'
      9↑S
ONCE UPON
```

If the left argument is a negative number, the elements are taken from the *end* of the vector.

```
      -3↑VECT
8 9 10
      -4↑S
TIME
```

If you take more elements than the vector has, the vector is “padded” (or filled) with zeros (for a numeric vector) or blanks (for a character vector).

```
      12↑VECT
1 2 3 4 5 6 7 8 9 10 0 0
      -12↑VECT
0 0 1 2 3 4 5 6 7 8 9 10
      -15↑'OH HO'
           OH HO
      ρ-15↑'OH HO'
15
```

## LOOPING

Looping is a useful technique to use in writing functions. Looping is merely a way to make the computer do the same thing over and over again. Consider the function *GUESS*:

```
      GUESS
I'M THINKING OF A NUMBER BETWEEN 1 AND 3.
WHAT IS IT?
ENTER YOUR NUMBER:
□:
      3
I'M SORRY.  THE NUMBER WAS 2.
TRY AGAIN?
YES
ENTER YOUR NUMBER:
□:
      3
RIGHT!  TRY AGAIN?
YES
ENTER YOUR NUMBER:
□:
      3
I'M SORRY.  THE NUMBER WAS 1.
TRY AGAIN?
NO
```

The computer repeated the section:

```
ENTER YOUR NUMBER:
□:
```

three times. It continues to repeat this section until you tell it to stop by entering *NO* in response to the question *TRY AGAIN?*

### branches

Looping is done in APL using the “branch” function. This function sends the computer to another line in the program. Branch has the form:

→4

### conditional branches

By itself, this function isn’t so useful. In the example above, it says “go to line [ 4 ] no matter what.” It is much more useful to use a *conditional* branch—for example, “go to line [ 4 ] only if a certain condition is met” (only if the user answers *YES*, for instance).

### Tests for Conditional Branches

To get a conditional branch, we use a test. This type of branch takes the form shown below.

→( 'Y' = 1 ↑ □ ) ρ 4

The test appears between the two parentheses so that it will be executed first. In the example above, the test checks to see whether the first letter of input (quote-quad input) is a *Y*. If it is, the answer is true, a 1:

→1 ρ 4

The expression  $1 \rho 4$  just gives the number 4 (it says, “give me one 4”), so we wind up with:

→4

and the computer goes to line [ 4 ]. If the user entered anything else (*NO*, for instance), the answer to the test would be false, a 0:

→0 ρ 4

Taking 0 of something (that is, reshaping it with 0) leaves a vector with no elements, therefore branching to no other location; the function continues on to the next line. In the case of *GUESS*, this caused the function to stop.

Let's define a function to see how this works. *FLIP* "flips a coin," checks to see if the answer is 1 or 2, and then displays *HEADS* or *TAILS* depending on the answer. It then gives you another chance. Enter:

```
      ▽ FLIP
[1]    N←2 5ρ'HEADSTAILS'
[2]    ((?2)=1 2)/[1]N
[3]    'TRY AGAIN? (YES OR NO): '
[4]    →('Y'=1↑⎕)ρ2
[5]    ▽
```

Let's try it:

```
      FLIP
HEADS
TRY AGAIN? (YES OR NO):
YES
TAILS
TRY AGAIN? (YES OR NO):
NO
```

The test in line [4] accepts quote-quad input and then checks to see if you entered the answer *YES* (or *YEAH* or *YUP*—the only requirement is that the first letter be a Y). If you enter *YES*, the computer goes back to line [2] and "flips the coin" again. If you enter *NO*, the program comes to an end.

## Line Labels

Using line labels is a much better way to do branching. The technique is simple—you just give names (labels) to certain lines of a function and then have the system branch to those lines according to their names. In other words,

```
      →LINE1
```

is just as good as:

```
      →1
```

as long as you've put the label *LINE1* on line [1] of your function. Using labels, the *FLIP* function might look like this:

```

▽ FLIP2
[1]  N←2 5ρ'HEADSTAILS'
[2]  MORE: ((?2)=1 2)/[1]N
[3]  'TRY AGAIN? (YES OR NO): '
[4]  →('Y'=1↑□)ρMORE
▽

```

The label is separated by a colon from the APL statement on the line. Of course, you can use any name you want for the label. You might prefer names that tell you what's happening—like *END*, *LOOP*, and so on. You can't have a label with the same name as a variable, though, so be sure to pick a unique name.

The great thing about labels is that we don't have to change our branches every time we edit the function. For example, if we decide to add a line to *FLIP2*, the present line [1] will become line [2]. So the function won't work right unless we change the branch part of the function. But if you use line labels, the branch will always go to the correct line, since it looks for the *name* instead of the line number.

Since line labels are so helpful, we'll use them in all our functions with branching from now on.

Now we're ready to define *GUESS*, a more complicated function. It includes two tests—one to check your number, and one to check for a *YES* or *NO* answer.

```

▽ GUESS
[1]  'I'VE PICKED A NUMBER BETWEEN 1 AND 3.'
[2]  'WHAT IS IT?'
[3]  LOOP: 'ENTER YOUR NUMBER:'
[4]  NUM←□
[5]  ANS←?3
[6]  →(ANS=NUM)ρOK
[7]  'I'M SORRY. THE NUMBER WAS ',(▼ANS),'. '
[8]  'TRY AGAIN?'
[9]  →('Y'=1↑□)ρLOOP
[10] →0
[11] OK: 'RIGHT! TRY AGAIN?'
[12] →('Y'=1↑□)ρLOOP
[13] ▽

```

---

Line [ 5 ] calculates the computer's answer by generating a random number between 1 and 3. Line [ 6 ] checks to see if the computer's answer and your guess are the same. If so, the computer goes to line [ 11 ], displays *RIGHT!*, and asks if you want to try again. If you say *YES*, the computer returns to line [3]. If you say *NO*, it stops.

If the test in line [ 6 ] shows that the answer and your guess are *not* equal, the computer continues to line [ 7 ], displays the correct answer, and asks if you want to try again. Line [ 9 ] uses the same test as line [ 12 ]. If your answer is *NO*, the computer goes to line [ 10 ], which says "go to 0." This tells the computer to *stop* (actually, to leave or exit from) the function.

**note**

Now that you've learned how to use loops, you may write a program with a so-called "endless loop" that you can't get out of. This will generally happen because you forget to include the test—which is the "exit" from the loop. If this happens, and you notice that the computer is doing the same thing for an overly long period of time, you can always interrupt the function. The computer will print the line number of the function where it stopped. You can then list the function and see if something's wrong. Remember to enter:

)SIC

before you edit the function, to avoid getting a *DEFN ERROR*.

**PROGRAMMING AIDS**

At this point, we'd like to introduce a few more tools to make your functions better. These tools are comments and local variables.

**Comments**

We use comments to describe what's happening in a function. A comment is merely a statement within a function that's not executed. It says something to the person reading the function rather than saying something to the computer. To tell the computer that what you are entering is a comment, you must precede the comment with a "lamp" symbol (⌘). It is called a lamp since it precedes the text that "illuminates" what is happening in the function. The following function includes a comment.

```
⌘ NOBLANK CHARVECT
[1] ⌘ REMOVES BLANKS FROM CHARACTER VECTOR
[2] (CHARVECT≠' ')/CHARVECT
[3] ⌘
```

**Local and  
Global  
Variables**

Let's define a function to compute sales data for a week given the units sold:

```
▽COMPUTESALES
[1]  ⌞ COLLECT UNITS SOLD, COMPUTE SALES
[2]  'ENTER SALES FOR THIS WEEK: '
[3]  SALES←⊞
[4]  'TOTAL SALES ',▽+/SALES×PRICE
▽
```

Let's create the price variable:

```
PRICE←10.30
```

To see what happens, let's create a *SALES* variable:

```
SALES←105
```

Now run the function:

```
COMPUTESALES
ENTER SALES FIGURES FOR THIS WEEK:
⊞:      175.24 159.4 211.35 201.56 213.45
TOTAL SALES 9898.3
```

Now check *SALES* again:

```
SALES
175.24 159.4 211.35 201.56 213.45
```

What happened? The function replaced our old variable *SALES* with the new figures. If you look at line [3] of *COMPUTESALES*, you'll see that our response is assigned to *SALES*, which "wipes out" whatever was in *SALES* before.



**local variables**

Local variables allow us to “lock” variables inside a function. We merely add the variable’s name to the function header, separated from it by a semicolon:

```
      ▼ COMPUTESALES
[6]   [0 0]
[0]   COMPUTESALES; SALES
[6]   ▼
```

Now let’s redefine *SALES*:

```
      SALES←105
```

Then run the function again:

```
      COMPUTESALES
ENTER SALES FIGURES FOR THIS WEEK:
□:      10 20 30 40 50
TOTAL SALES 1545
```

Check the variable *SALES* again:

```
      SALES
105
```

The function didn’t change the number in *SALES* this time.

**global variables**

The technique of using local variables is useful in protecting variables you’ve created in the workspace. (We call the general workspace variables “global variables” to distinguish them from local variables, which are meaningful only in the context of a particular function.) The technique also allows you to use the same variable names in many functions. For example, a sales system might contain many variables called *SALES*—one main (global) variable and lots of minor (local) ones locked inside functions.

Last but not least, local variables save room in the workspace, because local variables are erased as soon as the function has finished running. To see how this works, let's erase *SALES*:

```
      )ERASE SALES
      SALES
VALUE ERROR
      SALES
      ^
```

We get a *VALUE ERROR*, since *SALES* no longer exists. Now run *COMPUTESALES*:

```
      COMPUTESALES
ENTER SALES FIGURES FOR THIS WEEK:
□:      25 35 45 55 65
TOTAL SALES 2317.5
```

Now look for *SALES*:

```
      SALES
VALUE ERROR
      SALES
      ^
```

Even after running *COMPUTESALES*, no *SALES* variable exists in the workspace.

## **FUNCTIONS AND FILES**

We can use our new-found skill with programming to build some useful functions to work with files.

Let's say you want to build a file containing your favorite recipes. Each component will contain a single recipe. We want to put the recipes in the file using a function.

**building a recipe**

Let's think about what we have to do. A recipe is a good example of a character matrix. For example:

```

      RECIPE←3 12ρ'1 CUP FLOUR PINCH SALT
1 EGG  '
      RECIPE
1 CUP FLOUR
PINCH SALT
1 EGG

```

It's too difficult to do a real recipe this way. It would be easier to have a function build the matrix line by line. The function would add lines using catenation:

```

      NEWLINE←1 12ρ'BAKING PWDR '
      RECIPE,[1]NEWLINE
1 CUP FLOUR
PINCH SALT
1 EGG
BAKING PWDR

```

**building recipes  
with a function**

The function needs to assemble a recipe like this, then put it in the file. Also, we have to build in a "keyword" like *END* or *STOP* so the function will know where the recipe ends.

Define the function:

```

▽ FILERECIPE;RECIPE;NEWLINE
[1]  ⌘ COLLECTS RECIPE AND PUTS IN FILE
[2]  'RECIPES' ⌘FTIE 9971
[3]  'ENTER YOUR RECIPE LINE BY LINE'
[4]  '(USE "END" TO STOP):'
[5]  RECIPE←60↑⌘
[6]  RECIPE← 1 60 ρRECIPE
[7]  L1:'NEXT LINE:'
[8]  NEWLINE←60↑⌘
[9]  →(⌘/'END'=3↑NEWLINE)ρL2
[10] RECIPE←RECIPE,[1]NEWLINE
[11] →L1
[12] L2:RECIPE ⌘FAPPEND 9971
[13] ⌘FUNTIE 9971
[14] 'DONE.  NEW RECIPE FILED.'
[15] ▽

```

This function ties the file for you, then collects the text for the recipe. The tie number should be one that you're unlikely to use to tie any other files you might be working with, since file tie numbers used at the same time must be unique.

Lines [ 5 ] and [ 6 ] gather the first line of the recipe and turn the line into a one-row matrix (so it can be catenated using matrix techniques). Lines [ 7 ] and [ 8 ] gather the next recipe line, and line [ 9 ] checks to see if the first three characters are the word *END*. If not, the computer goes on to line [ 10 ], which catenates the line to the end of the recipe. Line [ 11 ] returns to line [ 7 ] for the next line of the recipe. When you enter *END*, the computer stores the recipe.

**making sure the  
lines are equal**

Remember, every line in the recipe must be of equal length for us to catenate them. We use "take" ( 60 ↑ ) to make sure each line is of equal length. "Take" pads each line of the recipe with blanks. When the function is finished, the recipe is a matrix with 60 columns and as many rows as you entered lines.

At the end of the function, the system unties the file again. Note that the function assumes that a file called *RECIPES* has *already* been created.

Let's try it. Create the file *RECIPES*:

```
'RECIPES' ⌞FCREATE 1
```

Then untie it:

```
⌞FUNTIE 1
```

Now run the function:

```
ENTER YOUR RECIPE LINE BY LINE  
(USE "END" TO STOP):
```

```
COOKIES
```

```
NEXT LINE:
```

```
(Just press RETURN or ENTER to get a blank line.)
```

```
NEXT LINE:
```

```
GO TO GROCERY STORE.
```

```
NEXT LINE:
```

```
BUY A BAG OF YOUR FAVORITE COOKIES.
```

```
NEXT LINE:
```

```
TAKE HOME, OPEN, AND EAT.
NEXT LINE:
END
DONE.  NEW RECIPE FILED.
```

We can read the recipes using `⎕FREAD`:

```
      'RECIPES' ⎕FTIE 1
      ⎕FREAD 1 1
COOKIES

GO TO GROCERY STORE.
BUY A BAG OF YOUR FAVORITE COOKIES.
TAKE HOME, OPEN, AND EAT.
```

Or, to make things friendlier, we could define a function `READ`:

```
      ▼ READ;ANS
[1]  ⌘ READS RECIPES STORED IN FILE
[2]  'RECIPES' ⎕FTIE 3371
[3]  'WHAT IS THE NUMBER OF THE RECIPE?'
[4]  ANS←⎕
[5]  ⎕FREAD 3371,ANS
[6]  ⎕FUNTIE 3371
[7]  ▼
```

```
      READ
WHAT IS THE NUMBER OF THE RECIPE?
⎕:
      1
COOKIES
```

```
GO TO GROCERY STORE.
BUY A BAG OF YOUR FAVORITE COOKIES.
TAKE HOME, OPEN, AND EAT.
```

### keeping a directory

We might not remember which component is which, but we can make `FILERECIPE` “smarter” so that it keeps a “directory” we can read. We can build a directory by taking the first line of each recipe (the title of the recipe) and storing it in a separate matrix in the file.

**using a driver function**

In fact, it might be better to build a system with small functions doing each job—tying the file, keeping the directory, and so on. We'll use a main program (also called a *driver function*) to run all the other ones.

The main program would look like so:

```

▽ RECIPES
[1]  A MANAGES RECIPE SYSTEM
[2]  TIEFILES
[3]  L0: 'LIST, READ, ADD, OR QUIT?'
[4]  '(L, R, A, OR Q):'
[5]  →('LRAQ'=1↑⍴)/L1,L2,L3,L4
[6]  'PLEASE ENTER L, R, A, OR Q.'
[7]  →L0
[8]  L1: LISTDIRECTORY
[9]  →L0
[10] L2: READREC
[11] →L0
[12] L3: ADDREC
[13] →L0
[14] L4: ⍴FUNTIE 1111 2222
[15] ▽

```

**branching to more than one line**

Notice that line [5] uses a new branching technique. The technique, which uses compression, allows the function to branch to different lines, depending on your response to the question on line [3]. If you enter *A*:

```

      'LRAQ' = 'A'
0 0 1 0
      0 0 1 0/L1,L2,L3,L4

```

The 1 in the compression vector matches up with the label *L3*, so the computer goes to line [12] and runs the *ADDREC* program. Then it returns to line [3] and asks you what you want to do now. Similarly, if you answer *L*, the computer goes to line [8], and so on. You're always returned to line [3] until you enter *Q* for *QUIT*.

**trapping errors**

There's one other new technique in this function. Line [6] reminds you that you must enter *L*, *R*, *A*, or *Q*. Since the test (line [5]) checks only for these four letters, any other response (like *X*, *Z*, or *G*) will cause the computer to go on to line [6], because we didn't tell it to go to [L1], [L2], [L3], or [L4]. So we catch ourselves in line [6], print a reminder, and go back to the beginning.

**tying files**

*TIEFILES*, the first part of the recipes system, is an easy one:

```
      ▼ TIEFILES
[1]   A TIES FILES FOR RECIPE SYSTEM
[2]   'RECIPES'  ⌵FTIE 1111
[3]   'DIRECTRY' ⌵FTIE 2222
[4]   ▼
```

We want two files so we can store the directory in a separate place. (This is *not* really efficient, but it has an advantage we'll see in a minute.)

Let's make a fresh start by erasing our old *RECIPES* file:

```
'RECIPES' ⌵FTIE 3371
'RECIPES' ⌵FERASE 3371
```

Now let's create a new set of files for our new system:

```
'RECIPES' ⌵FCREATE 1
'DIRECTRY' ⌵FCREATE 2
```

(We've deliberately spelled the name of the second file as *DIRECTRY* to keep it to the 8-character limit of some APL systems.) Since *TIEFILES* will tie these for us, let's untie them now:

```
⌵FUNTIE ⌵FNUMS
```

**listing the  
directory**

*LISTDIRECTORY* comes next. Enter:

```

      ▼ LISTDIRECTORY;NUMS;DIR
[1]  A PRINTS DIRECTORY FOR RECIPES SYSTEM
[2]  DIR←⊖FREAD 2222 1
[3]  NUMS←1 1↑ρDIR
[4]  'Q< >I5,55A1' ⊖FMT (NUMS;DIR )
[5]  ▼

```

The expression on line [3] figures out the number of rows in the directory (1↑ρDIR—the first number in the shape of the matrix), and then strings it out (using 1). Line [4] then formats the numbers next to the directory entries. The Q< > part is necessary if we want to have spaces displayed between the numbers and the entries.

It's easy to see why it's important to have a separate file for the directory—that way the recipe numbers start with 1 (that is, they start in component 1 of the *RECIPES* file).

**reading recipes**

*READREC* comes next. It assumes that you have listed the directory and know the number of the recipe you want.

```

      ▼ READREC;RECNO
[1]  A DISPLAYS A RECIPE BY NUMBER
[2]  'READ WHICH RECIPE (BY NUMBER): '
[3]  RECNO←⊖
[4]  ⊖FREAD 1111,RECNO
[5]  ▼

```

**adding recipes**

*ADDREC* comes last. It's very similar to *FILERECIPE*, except that it uses 55 columns for the recipe lines instead of 60. Enter:

```

      ▼ ADDREC;RECIPE;NEWLINE;DIR;NEWENTRY
[1]  A COLLECTS AND FILES RECIPE
[2]  'ENTER YOUR RECIPE LINE BY LINE'
[3]  '(USE "END" TO STOP): '
[4]  RECIPE←55↑⊖
[5]  RECIPE← 1 55 ρRECIPE
[6]  L1:'NEXT LINE:'
[7]  NEWLINE←55↑⊖
[8]  →(⌵/'END'=3↑NEWLINE)ρL2
[9]  RECIPE←RECIPE,[1]NEWLINE

```

---



```
[10]  →L1
[11]  L2:RECIPE ⍵FAPPEND 1111
[12]  DIR←⍵FREAD 2222 1
[13]  NEWENTRY← RECIPE [1;]
[14]  DIR←DIR,[1]NEWENTRY
[15]  DIR ⍵FREPLACE 2222 1
[16]  'NEW RECIPE FILED.'
[17]  ▽
```

Just like in *FILERECIPE*, we continue to add lines until the user enters *END*. Then the recipe is put in the file.

### updating the directory

Lines [12] through [15] keep the directory up-to-date by reading in the old directory, getting the new title, and adding it to the bottom of the directory matrix. The expression *NEWENTRY←RECIPE[1;]* on line [13] takes the first row of the recipe—the title. Line [15] returns the new, updated directory to the file.

### starting the system

To get this system started, we have to put something in the *DIRECTRY* file, so that *ADDREC* will find something there when it reads component 1 (line [12] of *ADDREC*). Otherwise, we'll get a *FILE INDEX ERROR*. However, we don't really want anything in the directory yet. So we "fool" APL by putting an empty matrix with 55 columns and *no* rows (*0 55ρ' '*) in the directory file.

Enter:

```
TIEFILES
BLANKROW←0 55ρ' '
BLANKROW ⍵FAPPEND 2222
⍵FUNTIE ⍵FNUMS
```

### running the system

Now we're ready to insert a recipe.

```
RECIPES
LIST, READ, ADD, OR QUIT?
(L, R, A, OR Q):
A
ENTER YOUR RECIPE LINE BY LINE
(USE "END" TO STOP):
GOAT STEW
```

```
NEXT LINE:
(Press RETURN or ENTER to get a blank line.)
NEXT LINE:
FIND LARGE GOAT.
NEXT LINE:
PUT IN LARGE KETTLE WITH CAMP AX.
NEXT LINE:
BOIL 12 HOURS.
NEXT LINE:
THROW AWAY GOAT AND EAT AX.
NEXT LINE:
END
NEW RECIPE FILED.
LIST, READ, ADD, OR QUIT?
(L, R, A, OR Q):
L
  1 GOAT STEW
LIST RECIPES, READ ONE, ADD NEW ONE, OR QUIT?
(L, R, A, OR Q):
R
READ WHICH RECIPE (BY NUMBER):
□:
  1
GOAT STEW

FIND LARGE GOAT.
PUT IN LARGE KETTLE WITH CAMP AX.
BOIL 12 HOURS.
THROW AWAY GOAT AND EAT AX.
LIST, READ, ADD, OR QUIT?
(L, R, A, OR Q):
Q
```

## SUMMARY

1. You can get the computer to collect numeric input by including a line like:

$A \leftarrow \square$

in your function. This is called “evaluated input” or “quad input.”

2. You can get the computer to collect character input by including a line like:

$A \leftarrow \square$

in your function. This is called “character input” or “quote-quad input.”

3. Catenation works with matrices, but you have to include the dimension (such as  $M1, [1] M2$ ) if you want the system to add *rows* rather than columns.
4. “Take” ( $\uparrow$ ) takes as many elements from a vector as you tell it to.
5. Looping is done using the “branch” function ( $\rightarrow$ ). Functions can branch to a line number or a line label, but labels are more meaningful, and they won’t change if line numbering changes.
6. Comments are included in a function to summarize in words what the function does. They must be preceded by a lamp symbol ( $\text{⌈}$ ) so that the computer will not interpret them as part of the function.
7. Local variables are variables “locked inside” functions. Using local variables keeps functions from changing your general workspace variables (global variables) and also saves room in the workspace.
8. You can use functions to collect data, store it in files, and retrieve it.

## EXERCISES

- A. 1. Write a new version of *DIE* (from Chapter 5) called *ASKDIE* that rolls the die, then gives you another chance. The function should produce the following display:

```
      ASKDIE
5
TRY AGAIN?  (YES OR NO):
YES
2
TRY AGAIN?  (YES OR NO):
NO
```

2. Write the function *ADDER*, which produces the following display:

```

      ADDER
ENTER NUMBERS TO BE ADDED:
□:
      107 542 122 889
THE TOTAL IS 1660.

```

3. Define a function called *ASKREPEAT* that repeats the character you enter. *ASKREPEAT* should produce the following display:

```

      ASKREPEAT
ENTER THING TO BE REPEATED:
*
REPEAT HOW MANY TIMES?
□:
      13
*****

```

4. Copy your *HIST* function (from Chapter 5) into the active workspace [use *)COPY MYWORK HIST*]. If you didn't save the function, copy it in from *LESSONS*. Use *HIST* inside a new function called *ASKHIST*. *ASKHIST* should produce the following display:

```

      ASKHIST
ENTER FOUR NUMBERS FOR BAR CHART:
□:
      6 12 3 1
□□□□□□
□□□□□□□□□□□□
□□□
□

```

- B. Design and create a system for collecting information on your household goods. The system should contain three main functions (*INVSTART*, *ENTER*, and *LIST*) and two subfunctions (*GETNAME* and *GETDETAILS*). (A subfunction is a function called by another function.) *INVSTART* merely ties the inventory file. *ENTER* collects data on your household inventory and files it. *LIST* reads in the data and lists it.

*LIST* is the longest function in the system, so we'll give it to you.

```
      ▽ LIST; DIR; NUMS; NUMBER; NAMES; NAM; DET
[11]  A MANAGES HOUSEHOLD INVENTORY SYSTEM
[12]  L1: 'DIRECTORY, ITEMS, OR QUIT? (D,I,Q):'
[13]  →('DIQ'=1↑⎕)/DI,ITEMS,EXIT
[14]  'PLEASE ENTER D, I, OR Q'
[15]  →L1
[16]  DI: DIR←⎕FREAD 1068 2
[17]  NUMS←⎕1↑⎕DIR
[18]  'Q< >I5,20A1' ⎕FMT (NUMS; DIR)
[19]  →L1
[20]  ITEMS: 'LIST WHICH ITEM (BY NUMBER)?'
[21]  NUMBER←⎕
[22]  NAMES←⎕FREAD 1068 2
[23]  DETAILS←⎕FREAD 1068 3
[24]  NAM←NAMES[NUMBER;]
[25]  DET←DETAILS[NUMBER;]
[26]  NAM,DET
[27]  →L1
[28]  EXIT:
      ▽
```

This function may look complicated, but all it does is read in and list either the directory or the detailed description of an item, similar to *RECIPES*. The only difference is that “individual items” are merely single lines (consisting of two parts—the item name and the item details) instead of an entire matrix.

Since the things we're interested in are single lines, we can store them in two matrices (one for names, one for details). These matrices are located in components 2 and 3 of the file *INVENTOR*. The function *LIST* uses the expressions *NAMES*[*NUMBER*;] and *DETAILS*[*NUMBER*;] to pull out the rows (item name and details) we want. Then it prints the two parts with the expression *NAM*, *DET*.

Notice that the *NAMES* matrix is also used for the directory. This is why we store the names and details in different components.

*ENTER* is a little different, since it calls two subfunctions, so we'll give that to you too.

---

```
▽ ENTER
[1] GETNAME
[2] GETDETAILS
[3] 'NEW ITEM FILED.'
▽
```

The functions left for you to do are *INVSTART*, *GETNAME*, and *GETDETAILS*. *GETNAME* and *GETDETAILS* work similarly to the first few lines of *FILERECIPE* and *ADDREC*. They read in what's in the file (names and details), gather a single line, add it to the bottom of the matrix, and replace the matrix in the file. The names should be 20 columns long (20 ↑ ▯) and the details 35 columns long. (Actually, you can use more columns, but our solution uses 20 and 35.)

A sample run might look like this:

```
INVSTART
ENTER
NAME OF ITEM:
19'' COLOR TV
SERIAL NUMBER, DATE PURCHASED, PRICE:
R8254471A    05/11/82    $421.63
NEW ITEM FILED.

LIST
DIRECTORY, ITEMS, OR QUIT? (D,I,Q):
D
1 19'' COLOR TV
DIRECTORY, ITEMS, OR QUIT? (D,I,Q):
I
LIST WHICH ITEM (BY NUMBER)?
▯:
1
19'' COLOR TV    R8254471A    05/11/82    $421.63
DIRECTORY, ITEMS, OR QUIT? (D,I,Q):
Q
```

Notice that, since the details are stored in one line, it is up to you to space out the three items properly. We left four spaces between the detail items, which works well.

Before running your system, be sure to start it up properly:

```
'INVENTOR' ⍺FCREATE 5
M←'FILE FOR STORING INVENTORY DATA. USE
<INVSTART>, <ENTER>, AND <LIST> WITH FILE.'
M ⍺FAPPEND 5
(0 20ρ' ') ⍺FAPPEND 5
(0 35ρ' ') ⍺FAPPEND 5
```

(These last two steps are necessary because *ENTER* will be looking for something in the file.)

**⍺FUNTIE 5**

There are lots of ways this system can be enhanced. For example, you can add items to the details list, such as place purchased, location in house, and so on. You can also store each detail in a separate component and have *LIST* ask you which details you want to see. Many variations are possible, but we'll leave it to you to tailor the system to your own needs and desires.

- C. Design a simple system to collect and format names and figures for a household (or any sort of a) budget. The function *BUDGET* collects and formats the data. The function *SHOWBUDGET* merely lists the data that *BUDGET* collected.

A run using *BUDGET* might look like so:

```
      BUDGET
ENTER BUDGET ITEMS ("END" TO STOP):
FOOD
NEXT ITEM:
RENT
NEXT ITEM:
GASOLINE
NEXT ITEM:
ENTERTAINMENT
NEXT ITEM:
END
ENTER BUDGET FIGURES FOR 4 ITEMS:
⍺:
      250 431.72 85 125
```

*BUDGET IS:*

|                      |          |          |
|----------------------|----------|----------|
| <i>FOOD</i>          | \$250.00 | 28.0•/•  |
| <i>RENT</i>          | \$431.72 | 48.4•/•  |
| <i>GASOLINE</i>      | \$85.00  | 9.5•/•   |
| <i>ENTERTAINMENT</i> | \$125.00 | 14.0•/•  |
| -----                |          |          |
| <i>TOTAL</i>         | \$891.72 | 100.0•/• |

*DON'T FORGET TO )SAVE IF THIS IS THE FINAL VERSION.*

The idea is that you can define as many different budgets as you like (by running *BUDGET* again), and the function will always tell you what the total budget amount is and what percentage each item is of the total. You keep defining new budgets until you find one that looks reasonable, then you enter *)SAVE*. Next time, you don't need to run *BUDGET*. You just run *SHOWBUDGET*, which uses the global variables created by *BUDGET*. *SHOWBUDGET* prints the same report as *BUDGET*, except that it omits the *BUDGET IS* and *DON'T FORGET* lines.

Here's *SHOWBUDGET*:

```

▽ SHOWBUDGET
[1]  A DISPLAYS BUDGET CREATED BY BUDGET FUNCTION.
[2]  ' '
[3]  FSB □FMT (INAMES;N;PERCENT)
[4]  44ρ'-'
[5]  FSB □FMT (T;+/N;100)
▽

```

This should give you some clues about how *BUDGET* works. *SHOWBUDGET* uses the same format string (*FSB*) that *BUDGET* does, and it uses *FSB* for the “meat” of the report as well as the total line. The underline is produced by *44ρ'-'*, and *T* is the word *TOTAL*, produced by the following sequence:

```

T←20↑'TOTAL'
T←1 20ρT

```



The variable  $N$  contains the budget numbers, and the expression  $+ / N$  finds the total budget figure. We “cheat” with the total percentage figure—we just use 100 instead of adding up the percentages. (The total will always be 99.9% or so, due to rounding.)

Otherwise, *BUDGET* is very much like the other data-gathering functions you’ve defined in this chapter. Two new tricks are included in it, though, which are explained below.

The line:

*ENTER BUDGET FIGURES FOR 4 ITEMS:*

is produced using the function line:

```
'ENTER BUDGET FIGURES FOR ', (⌈1↑ρINAMES), '
ITEMS: '
```

The expression in parentheses takes the first number of the shape of *INAMES* (item names), just like we did to produce the directory in Exercise B. Then it turns the number (the number of rows in *INAMES*) into character data.

The percentage figures are found by totaling the budget figures ( $TOT \leftarrow + / N$ ) and then dividing each figure by the total and multiplying by 100:

$$PERCENT \leftarrow 100 \times N \div TOT$$

## Chapter 10

### Exploring APL

---

In this chapter, you will learn about APL functions that sort data, manipulate data in a variety of ways, and perform math functions—from rounding numbers to solving simultaneous equations. Unlike the rest of the book, this chapter need not be read straight through; you can browse through it and read only the sections in which you are interested.

Up to this chapter, this book has attempted to build your knowledge of APL step-by-step, introducing new functions along the way to help solve new problems. The final objective was to be able to build applications such as those illustrated in Chapter 9, using a set of user-written functions incorporating files, formatting, and many primitive APL functions.

Chapter 10 is a little different. It's an “APL potpourri,” introducing APL functions that perform many standard computing tasks—such as sorting data, rounding numbers, locating data, and performing various mathematical functions. This chapter is meant to give you a taste of some of the more advanced things APL can do, giving you some useful tools and also whetting your appetite for other, more detailed APL textbooks.

Rather than read this chapter straight through, you may want to use the keywords in the margins to find the topics you are interested in, and read only those sections.

#### **SORTING**

Sorting is done in APL using the functions “grade up” (↑) and “grade down” (↓). Grade up and grade down return the *indices* required to sort the data in ascending or descending order. For example,

```
      ↑ 62 80 47 24
4 3 1 2
      ↓ 62 80 47 24
2 1 3 4
```

The first example says, “To sort in ascending order, we need the fourth element (24) *first*, the third element (47) *second*, and so on.” Thus, to sort the data, we have to use grade up/down along with the index function:

```
DATA←62 80 47 24
↑DATA
4 3 1 2

DATA[↑DATA]
24 47 62 80

DATA[↓DATA]
80 62 47 24
```

When two identical values are found in the data, the left-most value is given the lower index. For example:

```
DATA←62 80 47 24 80
↑DATA
4 3 1 2 5

↓DATA
2 5 1 3 4

DATA[↑DATA]
24 47 62 80 80

DATA[↓DATA]
80 80 62 47 24
```

If you sort a matrix, the grade functions sort along the first dimension. They treat the first column of the table as a vector and sort *it*, moving to the second column only if there are duplicates in the first column.

```
DATA2←4 2p54 100 22 23 15 49 62 99
DATA2

54 100
22 23
15 49
62 99
```

```

      ▼ DATA2
4 1 2 3

      ▲ DATA2
3 2 1 4

      DATA2[▲DATA2;]
22    23
22    49
54    100
62    99

```

The following example shows how to reorder a matrix based upon the significance of the columns.

```

      DATA←5 3ρ2 7 5 5 2 1 7 7 9 1 1 5 1 1 4
      DATA
2 7 5
5 2 1
7 7 9
1 1 5
1 1 4

```

To reorder using the second column as the most significant, the first as the next, and the third as the least significant:

```

      ▲DATA[;2 1 3]
5 4 2 1 3

      DATA[▲DATA[;2 1 3];]
1 1 4
1 1 5
5 2 1
2 7 5
7 7 9

```

To sort character data, you must use a sorting sequence as a left argument to ▲ or ▼. A convenient sorting sequence to use is ⎕AV, the “atomic vector,” which contains all possible characters used by the APL\*PLUS System. For example:

```
      NAMES←4 4ρ'JIM JUNEJOHNJOAN'
      NAMES
JIM
JUNE
JOHN
JOAN

      ⍳AV↑NAMES
1 4 3 2

      NAMES[⍳AV↑NAMES;]
JIM
JOAN
JOHN
JUNE
```

Also, a vector containing the alphabet can be used in place of `⍳AV`.

## ROUNDING

To do rounding, we use the APL functions “floor” (`⌊`) and “ceiling” (`⌈`). These functions give back either the next *lowest* whole number (floor) or the next *highest* whole number (ceiling). For example,

```
      ⌊4.1 4.9 -1.2
4 4 -2

      ⌈4.1 4.9 -1.2
5 5 -1
```

As you can see, however, this is not exactly rounding. With rounding, we want 4.1 to round *down* to 4, and 4.9 to round *up* to 5. So we use a little formula:

```
      ⌊.5 + 4.1 4.9 -1.2
4 5 -1
```

The addition of .5 pushes 4.9 up above 5, so `⌊` correctly rounds it to 5.

## LOCATING DATA

Often, you are interested in whether something exists (“Did I have any sales to customer 515 last month?”). If it exists, you would like to find out where it is (“Which salesman made that sale?”). We looked at one way of answering these questions in Chapter 4, but we will examine some other ways of answering them in this section.

## Membership

When we just want to know whether something exists or not, we can use the function “membership” ( $\epsilon$ ), sometimes called “element of.”

```
1      5  $\epsilon$  1 2 3 4 5 6
```

This statement asks the question, “Is 5 an element of the series 1 through 6?”

```
0      5  $\epsilon$  0 6 9 9
```

This statement asks the question, “Is 5 an element of 0 6 9 9?”

The answer is either true (1) or false (0).

```
1 1 0      5 6 11  $\epsilon$  1 1 0
```

The statement above asks the question, “Is 5 an element of 1 1 0, is 6 an element of 1 1 0, and is 11 an element of 1 1 0?”

The following statement is a convenient method of determining if all values of one object are contained in another object:

$$\wedge / DATA1 \epsilon DATA2$$

A 1 is returned if the left argument is a subset of the right argument, otherwise a 0 is returned.

```
1       $\wedge /$  1 9 8  $\epsilon$  1 1 0
```

## Where

Having discovered that something exists, we might want to know *where* it is located in our data. We can use the function “where” ( $\uparrow$  used dyadically) to do this. For example,

```
3      2 6 5 9  $\uparrow$  5
```

5            2 6 5 9 1 4 4

In the first example, 5 is located in the *third* position of the data (the left argument). In the second example, 44 is located *nowhere* in the data, so the answer is the index of the last element plus 1.

### Table Lookup

Finding the location of data in a *matrix* is a common problem, and this task is often called “table lookup.” Let’s say we have a table of names:

```
      NAMES←3 4ρ 'BILLJOHNANNA'
      NAMES
BILL
JOHN
ANNA
```

We have a new name that we want to add to the list, but only if it is not already in the table.

```
      NEWNAME←' JUNE '
```

We use the APL table lookup formula:

```
      NAMES^.=NEWNAME
0 0 0
```

The result, 0 0 0, shows that the *NEWNAME* is not in any of the rows of the table. Therefore we can append *JUNE* to the matrix of names:

```
      NAMES←NAMES,[1]NEWNAME
```

On the other hand,

```
      NAMES^.=' JOHN '
0 1 0 0
```

shows that *JOHN* is in the *second* row.

An important thing to notice about this formula is that the lengths must be correct—the vector must have the same number of elements as the matrix has columns. Otherwise, you get an error:

```

      NAMES^.= 'BOB'
LENGTH ERROR
      NAMES^.= 'BOB'
      ^
      NAMES^.= 'BOB'
0 0 0 0

```

A convenient method of checking if the *NEWNAME* has the correct length and to pad with spaces if it's too short, is:

```

      NEWNAME←'BOB'
      ρNEWNAME
3
      (⌈1↑ρNAMES)↑NEWNAME
BOB
      ρ(⌈1↑ρNAMES)↑NEWNAME
4
      NAMES^.= (⌈1↑ρNAMES)↑NEWNAME
0 0 0 0

```

As a final point, we should note that the table lookup formula is one example of using the APL “inner product” operator. Inner product allows you to combine two APL functions (in this case,  $\wedge$  and  $=$ ) in very powerful ways. We will see another example of inner product in the mathematics section of this chapter, under the heading of “Matrix Multiplication.”

## CONVERTING CHARACTERS TO NUMBERS

We’ve learned how to convert numbers to characters using the format function  $\text{⍒}$  (see Chapter 3). But what about going in the other direction, from characters to numbers?

APL provides a function called “execute,” which uses the  $\text{⍎}$  symbol to do this.

```

      ⍎ '59 72 63'
59 72 63
      5 + ⍎ '59 72 63'
64 77 68

```



Actually, execute is a more general function than we've indicated: it executes *any* character string that looks like a valid APL expression.

```

      * 'MAT←2 3ρ16'
MAT
1 2 3
4 5 6

```

This generality gets us in trouble when the character string contains something other than numbers and valid variable or function names.

```

      * '35 92 HI THERE'
SYNTAX ERROR
      * 35 92 HI THERE
      ^

```

The APL\*PLUS System provides a system function, `⍒FI` (format inverse), which is better to use for our specific need. `⍒FI` converts valid numbers, and changes invalid 1s to 0.

```

      ⍒FI '22 98.3 HI THERE'
22 98.3 0 0

```

Use  
45/0

The APL\*PLUS System also provides a system function `⍒VI` (verify input) to tell you what the character string contains. It returns 1s for valid numbers and 0s for anything else.

```

      ⍒VI '12.1 HI 52 ♣'
1 0 1 0

```

These two functions (`⍒VI` and `⍒FI`) used together allow you to collect numeric input using `⍒` rather than `⍒`. You can then use `⍒VI` to verify that the user had entered valid numbers, and if so, use `⍒FI` to convert them to numbers for further processing.

For example, to ignore the character data and to extract just the valid numeric data:

```

      DATA←'12.1 HI 52 ♣'
      (⍒VI DATA)/⍒FI DATA
12.1 52

```

This expression converts the data to numerics using `⎕F I`, and then selects only the valid numbers using `⎕V I` as a mark to compress 0s.

## SCALAR, VECTOR, OR MATRIX?

Sometimes you need to know whether the data you're dealing with is a scalar, vector, or a matrix. For example, some functions (such as "count") don't work with anything but scalars, and other functions (such as "indexing") work with everything *except* scalars.

## Rank

APL uses the concept of "rank" to deal with this need. Rank is defined as the number of dimensions your data has. The formula for finding rank is `ρ ρ`—the shape of the shape.

The shape of a vector is a single number showing how many elements the vector contains.

```

      ρ 5 8 10
3

```

So the rank of a vector is 1:

```

      ρ ρ 5 8 10
1
      ρ ρ 12 12.3 205 49 37
1

```

The shape of a matrix has two numbers in it, one for the rows and one for the columns:

```

      MAT←2 4 ρ 3 2 1 0
      MAT
3 2 1 0
3 2 1 0

      ρ MAT
2 4

```

So the rank of a matrix is 2:

```

      ρ ρ MAT
2

```

```
2      pp10 20p'HELLO THERE'
```

The rank of a scalar? Remember, a scalar has *no* shape (or, to be precise, a scalar's shape is the empty vector `⍬`):

```
      p5  
(six-space indent)
```

So a scalar's rank is 0:

```
0      pp 5
```

Understanding rank becomes simple when you think of it in terms of basic geometry. A scalar corresponds to a point; a vector corresponds to a line; and a matrix corresponds to a plane. Like a point, a scalar has a position but no length or depth; a line has length but no depth; a plane has both length and depth. Thus, a scalar has zero dimensions, a vector has one dimension, and a matrix has two dimensions.

Since APL allows data with a rank of 2, you might imagine that APL also allows data with rank 3, 4, or even higher. We call data with more than two dimensions “multidimensional objects.”

APL also allows data with a shape of 0, the empty vector (`⍬`), which has a rank of 1.

```
1      pp⍬
```

## Multi-dimensional Objects

One of the beauties of APL is that it allows you to deal with multidimensional objects as easily as scalars. You can build multidimensional objects with the reshape function, much as you build matrices:

```

      THREE←2 4 4ρ1 2 3 4
      THREE
1 2 3 4
1 2 3 4
1 2 3 4
1 2 3 4

1 2 3 4
1 2 3 4
1 2 3 4
1 2 3 4

```

Multidimensional objects are represented in APL by sets of tables. In the example above, we asked for 2 planes, 4 rows, and 4 columns. Above three dimensions, the geometry analogy breaks down, since it's hard to imagine objects with five or six dimensions. But rank will always tell you the number of dimensions:

```

      ρρTHREE
3

      ρρ3 4 5 2 4ρ0 1 2 3
5

```

**MANIPULATING DATA** This section covers a number of functions you can use to manipulate data: converting it to a vector, flipping it around, reversing its order, and so on.

**Ravel** Sometimes you need to convert data to a vector, because it can be simpler to work with a vector than with a matrix or a scalar. The function “ravel,” which uses the comma (,) symbol monadically, does this.

```

      MAT←2 3ρ16+16
      MAT
17 18 19
20 21 22

      ,MAT
17 18 19 20 21 22

```

You can see where ravel gets its name—it “ravels out” the matrix. Ravel also turns a scalar into a one-element vector:

```

      ρ 5
    , 5
5      ρ , 5
1

```

As the example shows, a scalar has no shape, but a raveled scalar has a shape of 1.

Ravel is useful when you want to ensure that the data you're using is a vector and *not* a scalar or matrix. If you're going to index into the data, for example, you must convert scalars to vectors, because indexing isn't defined for scalars. Also, `⎕SS` (string search), a very powerful system function described in the final section of this chapter, is defined for use only on vectors.

## Rotate

The “rotate” function does exactly what you'd expect it to—it rotates the data. Rotate uses the  $\Phi$  symbol.

When used monadically, rotate reverses the data completely:

```

      Φ 1 4
4 3 2 1

      MAT←3 4 ρ 'ABCDEFGH IJKL'
      MAT
ABCD
EFGH
IJKL

```

With a matrix,  $\Phi$  flips the columns within a row:

```

      Φ MAT
DCBA
HGFE
LKJI

```

Rotate can also be used dyadically. For example to rotate a vector by 2:

```

      2 Φ 4 5 6 7 8
6 7 8 4 5

```

Or by 1:

```
      1⌘4 5 6 7 8
5 6 7 8 4
```

You can also rotate in the opposite direction:

```
      -1⌘4 5 6 7 8
8 4 5 6 7
```

When used with a matrix, it rotates the individual columns or rows:

```
      MAT←3 4⌘'ABCDEFGH IJKL'
      MAT
ABCD
EFGH
IJKL
      1 2 1⌘MAT
BCDA
GHEF
JKLI
```

We rotated the first row by 1, the second by 2, and the third by 1. To rotate *columns* instead of rows:

```
      2 3 2 -1⌘[1]MAT
IBKL
AFCD
EJGH
```

We rotated the second column by 3, which brought it back around to its starting position.

## Overturn

You can use “overturn” as a synonym for  $\phi[1]$  (rotate across the first dimension). It uses the  $\ominus$  symbol.

When used monadically, overturn reverses the rows completely:

```
      ⊖MAT
IJKL
EFGH
ABCD
```

Overturn can also be used dyadically:

```
      2 3 2 -1 ⌘ MAT
IBKL
AFCD
EJGH
```

We rotated the first column by 2, the second column by 3, the third column by 2, and the fourth column by -1.

## Transpose

A close relative of rotate and overturn is “transpose,” which uses the  $\mathfrak{D}$  symbol. Transpose allows you to shift the *dimensions* of your data, turning the rows dimension into the columns dimension, for example:

```
      ρ MAT
3 4      ⍷ MAT
AEI
BFJ
CGK
DHL
4 3      ρ ⍷ MAT
```

When used dyadically:

```
      2 1 ⍷ MAT
AEI
BFJ
CGK
DHL
```

This expression tells APL to make the first dimension the second, and vice versa. So what were the rows (such as *ABCD*) are now columns. The transpose function might be useful if you want to sort the *columns* of a matrix rather than the rows [remember that the APL sorting functions ( $\blacktriangle$  and  $\blacktriangledown$ ) sort the *rows* of a matrix].

## GRABBING PIECES OF DATA

We've already learned that "take" can be used to grab pieces of data (see Chapter 9). The take function grabs as many elements as you specify:

```
2↑6 10 12 14
6 10
```

```
-2↑'ABCDEF'
EF
```

```
2 2↑MAT
AB
EF
```

## Drop

The "drop" function, which uses the ↓ symbol, does the opposite of take. It drops off as much data as you specify:

```
2↓15
3 4 5
```

```
-2↓15
1 2 3
```

Drop can be tricky with matrices, since it drops both rows *and* columns:

```
      MAT
ABCD
EFGH
IJKL

2 1↓MAT
JKL
```

We dropped 2 rows, leaving only the last row, and we also dropped the first column. To eliminate only the first row, but leave all columns, use a 0 drop:

```
1 0↓MAT
EFGH
IJKL
```

It can be very helpful to use take and drop together to move through your data, processing individual chunks of it.



```
DATA←'PIECE1PIECE2PIECE3'
```

First you use take to grab the first chunk:

```
6↑DATA  
PIECE1
```

Once you've finished with the first chunk, you drop it off, then grab the next chunk:

```
6↑6↓DATA  
PIECE2
```

## ASSORTED MATH FUNCTIONS

This section deals with various math functions available in APL. We'll introduce everything from exponentiation and logarithms to matrix multiplication and the solution of simultaneous equations.

### Exponentiation

The "power" function gives you the ability to work with exponents. It uses the  $\star$  symbol.

```
2★0 1 2 3 4  
1 2 4 8 16
```

### Logarithms

The "log" function uses the  $\odot$  symbol. It returns the natural logarithm.

```
○1 10  
0 2.302585093
```

Used dyadically, you can specify the base for the logarithm.

```
2 4○8 0.25  
3 -1
```

The result tells us that 2 to the third power is 8, and 4 to the  $-1$  power is 0.25.

### Outer Product

Outer product uses an APL operator,  $\circ$  (pronounced "jot dot" or "null dot"), to combine *each* element of the left argument with each element of the right argument. For instance, an easy way to produce a multiplication table is:

```

      1 2 3 •. × 1 2 3 4
1 2 3 4
2 4 6 8
3 6 9 12

```

The first row of the result shows 1 times 1 2 3 4, the second row shows 2 times 1 2 3 4, and so on.

Any scalar dyadic function can be used with outer product:

```

      1 2 3 •. ≠ 1 2 3
0 1 1
1 0 1
1 1 0

```

```

      1 2 3 •. ≥ 1 2 3
1 0 0
1 1 0
1 1 1

```

## Inner Product

Inner product is an even more powerful operation. Inner product takes *two* scalar dyadic functions as its arguments, one on each side of the dot ( . ). It uses the function on the right to combine each element of the left argument with each element of the right argument (similar to outer product). But then it uses the function on the left to do a “reduction” (such as a “plus reduction” or a “max reduction”—see Chapter 3) on the result.

We’ve already looked at one example of inner product,  $\wedge . =$  (see the Locating Data section in this chapter).

## Matrix Multiplication

Another example is to use  $+$  and  $\times$  to perform matrix multiplication:

```

      M1 ← 2 3 ρ 1 6
      M1
1 2 3
4 5 6

```

```
      M2←⊖M1
      M2
1  4
2  5
3  6

      M1+.  
×M2
14 32
32 77
```

Notice that the left argument must have the same number of rows as the right argument has columns. The result then has the same number of rows as 1, but the same number of columns as 2.

Any of the scalar dyadic functions can be used as either the left or right functions in inner product, making this a *very* powerful operation.

## Matrix Division

APL also provides a function to solve matrix division problems. Given a matrix *M* and a vector *R* (with the same number of elements as *M* has rows), the expression *R*⍷*M* will solve the equation *MX*=*R*, where *X* is a vector of unknowns. The “matrix divide” function uses the ⍷ symbol.

```
      M←3 3⍴1 1 1 0 1 1 0 0 1
      M
1 1 1
0 1 1
0 0 1

      2 3 5⍷M
-1 -2 5
```

If the equation has more than one solution, or if matrix *M* is singular, ⍷ returns a *DOMAIN ERROR*.

### Least-Squares Approximation

If  $M$  has more rows than columns,  $R \leftarrow M$  will provide a least-squares approximation.

```

      M←3 2ρ1 0 0 1 0 0
      M2
1 0
0 1
0 0

      1 1 1⊖M2
1 1

```

Dyadic  $\ominus$  can be used to solve linear equations, to fit a straight line, and to fit a polynomial curve. For example, consider the following set of 3 linear equations:

$$\begin{aligned} x + y + 2z &= 9 \\ 2x + 4y - 3z &= 1 \\ 3x + 6y - 5z &= 0 \end{aligned}$$

This would be represented in APL by two matrices

```

      A←3 3ρ1 1 2 2 4 -3 3 6 -5
      A
1 1 2
2 4 -3
3 6 -5

      B←3 1ρ9 1 0
      B
9
1
0

```

where  $A$  represents the coefficients of the linear equations and  $B$  represents the resultants.

The solutions for  $x$ ,  $y$ , and  $z$  in the linear equations can be determined by

$$B \mathbb{A}$$

$$\begin{matrix} 1 \\ 2 \\ 3 \end{matrix}$$

where  $x = 1$ ,  $y = 2$ , and  $z = 3$ .

### Matrix Inversion

Used monadically,  $\mathbb{M}$  returns the inverse of matrix  $M$ , provided that  $M$  is square and nonsingular. If  $M$  has more rows than columns, a least-squares approximation to the right inverse of  $M$  is returned.

$$\mathbb{M} \begin{pmatrix} 2 & 2 & 1 & 1 & 0 & 1 \\ 1 & -1 & 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix}$$

### Combinations

The “combinations” function is used to find the number of distinct groups of a particular size that can be selected from a group of equal or larger size. Such problems are usually described as “five choose two,” “four choose three,” and so on. Combinations uses the “shriek” symbol,  $!$ .

$$\begin{matrix} & 1 & 2 & 5! & 5 & 4 & 5 \\ 5 & 6 & 1 \end{matrix}$$

$$\begin{matrix} & 0 & 1 & 2 & 3! & 3 \\ 1 & 3 & 3 & 1 \end{matrix}$$

### TRIGONO- METRIC FUNCTIONS

APL provides a full set of functions to find values for trigonometric analysis. All are based around the “circular” function, which uses the  $\circ$  symbol. Used monadically,  $\circ$  is defined as “pi times  $x$ .”

$$\begin{matrix} \circ 1 \\ 3.141592654 \end{matrix}$$

The circular function is used dyadically with a set of numeric codes as its left argument, ranging from 1 to 7 and from  $-1$  to  $-7$ . For example,  $1 \circ X$  returns the sine of  $X$ , and  $-1 \circ X$  returns the cosine;  $3 \circ X$  returns the tangent;  $-3 \circ X$  the arctangent; and so on.  $X$  is given in radians, which can be found using the formula:  $\circ \text{ DEGREES} \div 180$ .

$-1 \circ X$  returns  
Arccos

See your system's reference card or reference manual for a complete list of the trigonometric functions.

## SOME APL IDIOMS

Even a language like APL, with its rich set of built-in functions, doesn't include a function for *every* need. However, you can put together a few APL functions and form short expressions that are so generally useful that they almost serve as primitive functions themselves. We call these short expressions "APL idioms."

Once you have used an idiom for a while, you learn to recognize it immediately, without having to decipher its individual parts. If you know a lot of common idioms, it's much easier to read someone else's APL programs, since you'll immediately recognize the purpose of many lines.

You've already learned a few idioms in the earlier chapters of this book. For example,

`(SENTENCE# ' ')/SENTENCE`

is the idiom for removing all blanks in a character vector. The expression for rank,

`ρ DATA`

is also an idiom. In fact, most of the expressions introduced in this chapter could be considered idioms. This section will introduce you to a few more useful idioms.

## Numeric or Character?

An idiom for finding out whether your data is numeric or not is

`0=1↑0ρ DATA`

*- PROTOTYPE less below*

For example,

1 `0=1↑0ρ 88 99 32` ← empty

0 `0=1↑0ρ 'HELLO THERE'`

*Watch!*

*empty - prototype ← ↑ empty*  
*0 ← numeric*  
*1 ← character*

Op anything is empty

The corresponding test for character data is

`' ' = 1 ↑ 0 ρ DATA`

**Branch  
to Next  
Line**

You've already learned the idiom for branching to another line in the program:

`→ (CONDITION) ρ LINE1`

For example,

`→ (NUM = 0) ρ OK`

An idiom for "go to the next line if the condition is met" is

`→ (CONDITION) ρ ⌊ LC + 1`

The system function `⌊ LC` is the line counter, and it gives you the line number of the user-defined function currently being executed. So if you go to `⌊ LC + 1`, you're going to the *next* line. An example:

```

.
.
.
[6] → (NUM > 0) ρ ⌊ LC + 1 ⋄ 'NONE FOUND.' ⋄ → EXIT ✓
[7] (▼ NUM), ' OCCURRENCES FOUND.'
.
.
.
[20] EXIT:
[21] ▼

```

If `NUM` isn't larger than 0, nothing was found, so a message is printed and we exit the program. Otherwise, we go to line 7 (`⌊ LC + 1`) and report on the number of occurrences found.

Cond'l  
Branching

Note:

$(2=3)/35 \rightarrow \text{blank (empty)}$   
 $(2=2)/35 = 1$

**Finding  
Indexes**

A common idiom for getting all the indexes of a vector is

`⍋V`

To illustrate:

```
V←8 17 -2
⍋V
3
⍋V
1 2 3
V[⍋V]
8 17 -2
```

**Locating  
Data Based  
on a Test**

This idiom (`⍋V`) is often used to form longer idioms. For example, the last example above shows that we get *all* of *V* if we index it with `⍋V`. But we can index *V* based on a test:

```
(V>0)/⍋V
1 2
V[(V>0)/⍋V]
8 17
```

The first statement says “Give me the indexes of all elements in *V* that are greater than zero.” The second statement says, “Index *V* with those indexes, giving me all the numbers in *V* greater than zero.”

**Replacing  
A with B in  
a Vector**

A related idiom allows you to replace all occurrences of 9 in vector *V* with 777. For example,

```
V←8 9 12 27 9
V[(V=9)/⍋V]←777
V
8 777 12 27 777
V←'MISSISSIPPI'
V[(V='S')/⍋V]←'F'
V
MIFFIFFIPPI
```



### Testing for an Empty Result

The idiom for checking for an empty vector is

$$0 = \rho A$$

This idiom is useful when accepting  $\square$  input, to check whether the user simply pressed ENTER instead of typing something.

A more general idiom for checking for an empty result is

$$0 \in \rho DATA$$

This idiom will work with data of any rank.

### Finding a Substring

The final idiom to be introduced here is *not* an idiom in the APL\*PLUS System, but rather a system function  $\square SS$  (for *string search*). It appears here because it is very useful, and also because it replaces long and complex idioms in other versions of APL.

To find substring *STRING* in the sentence *TEXT*, use the expression:

```
TEXT  $\square SS$  'STRING'
```

A string of 0s and 1s is the result, with a 1 marking the beginning of the substring.

```
TEXT ← 'I YAM WHAT I YAM.'
TEXT  $\square SS$  'YAM'
```

```
0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
```

If we're interested simply in whether the string is in the text or not:

```
1  $\in$  TEXT  $\square SS$  'YAM'
```

```
1
```

Where does the string start in the text?

```
(TEXT  $\square SS$  'YAM') / 1  $\rho$  TEXT
```

```
3 14
```

This section introduced only a few of the common APL idioms. More idioms can be found in any of the standard APL texts, such as *APL:An Interactive Approach*. Perhaps the most complete collection of APL idioms is the *FINNAPL Idiom Library*, available from the Finnish APL Association.

## Appendix A

### Solutions to Exercises

---

#### Chapter 2 A.

1. 4 5 6

2. 3

3.  $-2$

4. 6 12

5. 6 12

This is the same answer as for Exercise 4, since 3 is multiplied by both 2 and 4 in both cases.

6. *SYNTAX ERROR*

This is because the plus function expects data on both sides. That is, plus is a dyadic function.

7. *SYNTAX ERROR*

Since the high minus is part of the number 2 ( $-2$  = negative two), this expression has no meaning. Use the “middle minus” ( $-$ ) to subtract; for example,  $3 - 2$ .

8. 5 2 3

9. *DOMAIN ERROR*

Division by zero is not defined in arithmetic. Thus, zero is outside of the domain of the division function.

10. 40

Remember to evaluate right-to-left. That is,  $2 + 6 = 8$ ,  $5 \times 8 = 40$ .

11. *LENGTH ERROR*

The two vectors are not of equal length, and thus cannot be added.

12. 1 6

Parentheses are used to change the order of execution.

13. *SYNTAX ERROR*

In regular math, parentheses can be used to show multiplication, but in APL, the times sign must be used.

14. 4 2

15. 8

16. 1

**B.**

1. 4

2. 3 6 9

3. 2 3 4  
5 6 7

4. 2 8 14

5. 0 2 4  
6 8 10

6. *RANK ERROR*

Vectors cannot be added to matrices.

7. 4 5 6  
7 8 9

8. 8 6 4

9. *LENGTH ERROR*

The vector on the left contains only two elements, but *B* contains three.

10. 2 8 14

C.

1. 3

2. 4

3. 7

Since APL evaluates right-to-left, it links the two vectors together and then takes the shape of the new, longer vector.

4. 0 0 1 5

A scalar is catenated to the end of a vector.

5. 4

The new vector has four elements.

6. 1 2 4 9

Since the new vector has four elements and V2 also has four elements, the two vectors can be added.

7. 4 4 4

Dyadic rho is used to create a vector. Instead of rows and columns, we get one string, with as many elements as the number on the left specifies.

8. 2 1 2

1 2 1

2 1 2

9. 0

0

Here, rho uses the contents of V3 (2 1) to reshape V into a matrix with two rows and one column. Since the 1 isn't required (V1 contains 0 0 1), APL just leaves it out.

10. 5 0 0 1

A 5 is catenated to the front of a vector.

If you missed Questions 4, 6, 8, and 9, relax. These questions were meant to point out some additional properties of reshape and catenate rather than to test you on your previous knowledge. Try some experiments of your own with these functions and try to interpret the results—the worst that could happen is that you'll receive an error message.

D.

1. `.20 .30 .50×12 6 2` or `12 6 2×.20 .30 .50`

2. `600 700 1000 950×.05 .0525 .055 .06`

3. `100×142 167 161 128÷175`

4. `2116+(7216-6000)×.25` or `2116+.25×7216-6000`

The second expression works because the difference (`7216-6000`) is taken first, the multiplication is done next (`.25×1216`), and then the addition (`2116+304`). Remember, think right-to-left!

5.

|                         |                                                       |
|-------------------------|-------------------------------------------------------|
|                         | <code>SALES←2 3ρ12 9 3 11 12 6</code>                 |
|                         | <code>PRICES←2 3ρ3.25 1.79 2.55 3.45 1.85 2.75</code> |
|                         | <code>SALES×PRICES</code>                             |
| <code>39</code>         | <code>16.11 7.65</code>                               |
| <code>37.95 22.2</code> | <code>16.5</code>                                     |

6.

|  |                                       |
|--|---------------------------------------|
|  | <code>OLDGRADES←GRADES</code>         |
|  | <code>GRADES←93 87 80 90 78 85</code> |

7.

|                                         |                                 |
|-----------------------------------------|---------------------------------|
|                                         | <code>INVCOST←QUANT×COST</code> |
|                                         | <code>INVCOST</code>            |
| <code>472.5 773.67 704.48 155.25</code> |                                 |

Chapter 3 A.

1.

|                  |                           |
|------------------|---------------------------|
|                  | <code>ARROWS←'↔↑↓'</code> |
|                  | <code>ARROWS</code>       |
| <code>↔↑↓</code> |                           |

2. `CHARS ←  
'"~<=>≠∨^→*oι↓↑~ρ∈ω?αΓL_∇Δ°' '□(  
)c▷∩∪⊥τ|;: \'`

Of course, the order will vary depending on how you decide to type in the symbols. The only thing you might have trouble with is the single quote. Remember you must have an even number of quotes in a line, so enter an extra quote if you run into a problem. Then reassign the variable, and remember to double the quote.

3. `NUMBERS←3 5ρ'ONE TWO THREE'  
NUMBERS  
ONE  
TWO  
THREE`

4. `ALPH←5 1ρ'ABCDE'`

5. The shape of our *CV* is 4, since we included four spaces. If you included a different number of spaces, the shape will also be different.

6. The shape is 8. Since APL translates the double quotes back to a single quote, the double quotes count as one element only.

7. `A[5 2 3]  
TAN`

8. `B,C  
BLUEBELL  
D,C  
JINGLEBELL  
C,E  
BELLRINGER`

9. a. `ALPH[20]  
T`  
b. `SYMB[4]  
*`  
c. `ALPH,SYMB  
ABCDEFGHIJKLMNOPQRSTUVWXYZ∇Δ|*°`

- d.             $\rho$ ALPH,SYMB  
      3 1
- e.            ALPH[1 12] or  
              ALPH[1 2 3 4 5 6 7 8 9 10 11 12] or even  
              12 $\rho$ ALPH
- f.            ALPH[?26 26 26] or  
              ALPH ?3 $\rho$ [26]

Try the expression a few times to see the result.

10.            10 $\rho$ '\*'  
              \*\*\*\*\*

You have asked APL to “reshape” the single star into a character vector with 10 stars.

11.            M←'THE NUMBER IS '  
              M,?20  
              THE NUMBER IS 20

12.            P1←'YOU HAVE SPENT '  
              P2←' DOLLARS.'  
              P1,(?10),P2  
              YOU HAVE SPENT 10 DOLLARS.

You need parentheses around the numeric part (?10) to keep concatenate from joining the character data in P2 to the number 10 before it's been converted to character data. If you forgot the parentheses, you probably got a *DOMAIN ERROR*, since concatenate is not defined for use with mixed character and numeric data.

- B.
1. 4 4 7 6
2. 3 1 4 4
3. 4 2 2 8
4. 2 1 2 2
5. 4 1 7 8



6. 4 1 7 8

For this function, it makes no difference which side of the function symbol the data appears on.

7. 3 1 2 6

8. 4 3 3  
9 7 3

9. 3 3 5  
6 8 3

10. 4 2 3  
4 4 1

11. 1 3 4  
4 4 1

12. 4 2 3  
9 7 1

You can use maximum to compare  $C$  to itself, but the result is merely  $C$  again.

13. 4 3 5  
9 8 1

14. 1 2 3  
6 7 1

C.

1. 17

2. 15

3. 9 17

4. 9 15

5. 8

---

6. 1

7. 4 9

8. 1 1

9. 13 9 4

10. 7 11 6

11. 26

12. 24

13. 9

14. 1

**D.**

1. `TRANSACT[3]`  
17

2. `+/TRANSACT`  
154.22

3. `+\TRANSACT`  
153.17 133.17 150.17 114.42 141.97 154.22

4. `TRANSACT[1]-TRANSACT[6]`  
140.92

If you got this one on your own, congratulations! It's a bit different than anything we did in the chapter.

5. `(+/TRANSACT)×.05` or `.05×+/TRANSACT`  
7.711 7.711

The first expression is closer to regular arithmetic, but the second one saves typing two parentheses by “thinking APL”—that is, right-to-left execution.

E.

1.  $\rho DEPOSITS$   
8

2.  $(+/DEPOSITS)-+/WITHDRAWALS$   
697.72

The parentheses on the left are required to make sure we get what we want—total withdrawals subtracted from total deposits. Without parentheses on the left, the total withdrawals would be subtracted from each deposit, then all these negative numbers would be summed up. Try it!

3.  $DEPOSITS \leftarrow DEPOSITS, 450$   
 $DEPOSITS$   
120 25 115 375 800.12 22 75 82.75 450

4.  $\lceil /DEPOSITS$   
800.12  
 $\lfloor /DEPOSITS$   
22

5.  $\lceil /WITHDRAWALS$   
372.12  
 $\lfloor /WITHDRAWALS$   
15

6.  $(\lceil /DEPOSITS) - \lceil /WITHDRAWALS$   
428

Don't forget the parentheses!

7.  $DEPOSITS[1\ 3\ 5]$   
120 115 800.12

8.  $WITHDRAWALS[1\ 5]$  or  
 $WITHDRAWALS[1\ 2\ 3\ 4\ 5]$  or even  
 $5\rho WITHDRAWALS$

**F.**

1.           Z  
      HOW  
      CAN  
      THAT  
      BE?
2.           Z[1;1 2 3]  
      HOW
3.           Z[3;2 3 4]  
      HAT
4.           Z[4;3]  
      ?
5.           Z[4;3 3 3] or  
              3ρZ[4;3]
6.           Z[?4;1]  
Enter the expression a few times to see what happens.
7.           Z[2;?4]  
Remember that you have a blank in this row—so the result may be “invisible” if the random number the system chooses is 4.

**Chapter 4 A.**

1.           + / ' E ' = ' TENNESSEE '  
          4
2.           + / TRANSACT < 0  
          3  
TRANSACT < 0 gives a result of 1s and 0s; the + / adds up the 1s, which correspond to negative numbers (that is, withdrawals).
3.           + / TRANSACT ≥ 0  
          4
4.           + / ' ' = S  
          9

5.  $+ / SALES \leq 375$   
5

B.

1.  $\wedge / ACCOUNTS \geq 0$   
0

The answer is no—some accounts must have negative balances.

2.  $+ / ACCOUNTS < 0$   
3

3.  $+ / ACCOUNTS \geq 0$   
7

4.  $\vee / ACCOUNTS > 200$   
1

The answer is yes—there must at least one account greater than \$200.

$+ / ACCOUNTS > 200$   
1

There is only one.

5.  $\vee / (ACCOUNTS > 120) \wedge ACCOUNTS < 175$   
1

Yes, there are some accounts between \$120 and \$175.

$+ / (ACCOUNTS > 120) \wedge ACCOUNTS < 175$   
2

In fact, there are two accounts between \$120 and \$175.

6.  $+ / (ACCOUNTS < 0) / ACCOUNTS$   
-97

$(ACCOUNTS < 0) /$  uses compression to pick out the negative balances, then the  $+ /$  adds them.

7.  $+ / (ACCOUNTS \geq 0) / ACCOUNTS$   
633

C.

1.  $(PRICES > 15) / PRICES$   
 30.29 18.95 19.95

Don't forget the parentheses around the expression on the left! Without the parentheses, you'll get either a *DOMAIN ERROR* or a *LENGTH ERROR*.

2.  $(PRICES \leq 3.07) / PRICES$   
 1.79 3.07 2.15

3.  $(PRICES = 17.75) / PRICES$

This expression gives no result, because none of the prices equals \$17.75.

4.  $((PRICES > 10) \wedge PRICES < 20) / PRICES$   
 10.77 18.95 19.95

You may have had trouble with all of the parentheses in this exercise. Remember that you must always have an *even* number of parentheses (just like quotes). If you omitted any parentheses, you probably got an error message.

5.  $((PRICES < 2) \vee PRICES > 30) / PRICES$   
 1.79 30.29

D.

1. 3  
 6  
 9

2. *OE*  
*OE*  
*OE*

3. *MOE*

4. 4 5 6

5. 6

The rightmost part ( $0 \ 1 \ 0 / [1] A$ ) chooses the middle row of *A*, then the  $0 \ 1$  part chooses the final element of that row.

6. *LENGTH ERROR*, since *A* has only three elements.

---

E.

```
1.      (CUSTNO=3) / [1] CORDERS
      72 35 99
```

The [1] part is necessary because we're looking for a *row* of orders. Remember, each row corresponds to one customer.

```
2.      (CUSTNO=4) / [1] CADDRESS
      27 MEMORIAL DR.
```

Again, we need a row of the address matrix.

```
3.      (MONTHS=11) / CORDERS
      4 12
      8 4
      3 5
      7 8
```

This time we need a *column*, because we're searching by month rather than by customer.

```
4.      (CUSTNO=1) / [1] (MONTHS=12) / CORDERS
      8 1 1
or
```

```
      (MONTHS=12) / (CUSTNO=1) / [1] CORDERS
```

Either expression will work, but you need to remember to use the [1] when compressing with *CUSTNO*. That's because *CUSTNO*'s length matches the number of *rows* in *CORDERS*. If you try *(CUSTNO=1) / CORDER*, you get a *LENGTH ERROR*, because you're trying to compress three columns using four 0s and 1s.

## Chapter 5

**Note:** You will probably have different names for the function arguments than the ones shown here.

A.

We show all of the steps in one continuous display.

```
      v NUMBERS
[6]   [2]
[2]   TOO
[3]   [4]
[4]   FORE
```

---

```

[5]      [3⍴0]
[3]      THREE
[3]      TREE
[4]      [1.5]
[1.5]    TWO
[1.6]    [~5]
[5]      []
          ▽ NUMBERS
[1]      ONE
[1.5]    TWO
[2]      TOO
[3]      TREE
[4]      FORE
          ▽
[5]      [0]
[0]      COUNT
[5]      []
          ▽ COUNT
[1]      ONE
[1.5]    TWO
[2]      TOO
[3]      TREE
[4]      FORE
          ▽
[5]      ▽

```

Don't forget to get out of function definition mode by typing a ▽.

**B.**

1.  $R \leftarrow A \text{ ADD } B$
2.  $R \leftarrow A + B$
3.
 

```

          ▽ R ← A ADD B
      [1]      R ← A + B
      [2]      ▽

          A ← 4 ADD 2
          A

      6

```
4.  $RUNTOT \text{ DATA}$



```
5.      +\DATA

6.      ▽RUNTOT DATA
[1]     +\DATA
[2]     ▽
        RUNTOT 10 20 30 40
10 30 60 100
```

C.

```
1.      ▽HI
[1]     'HELLO.  MY NAME''S APL.'
[2]     'WHAT''S YOURS?'
[3]     ▽
```

Don't forget to double the quotes.

```
2.      ▽HALF NUM
[1]     NUM×.5
[2]     ▽
        HALF 3.14159
1.570795
```

```
3.      ▽HALF
[2]     [0]
[0]     R←HALF NUM
[2]     [1]
[1]     R←NUM×.5
[2]     [0]
        ▽R←HALF NUM
[1]     R←NUM×.5
        ▽
[2]     ▽
```

You *must* remember to assign a result both in the header *and* in the body of the function. If you don't, you'll get a *VALUE ERROR* in Exercise 4.

```
4.      10+HALF 3.14159
11.5070795
```

```

5.      ∇ROWTOT MAT
      [1]  +/MAT
      [2]  ∇
           ROWTOT MAT
      3 7 11

6.      ∇COLTOT MAT
      [1]  +/[1]MAT
      [2]  ∇
           COLTOT MAT
      9 12

7.      ROWTOT MAT2
      50 66 82
           COLTOT MAT2
      45 48 51 54

8.      ∇RPT DATA
      [1]  DATA
      [2]  '-----'
      [3]  COLTOT DATA
      [4]  ∇

           RPT DATA
      11 12 13 14
      15 16 17 18
      10 20 21 22
      -----
      45 48 51 54

```

**D.**

```

1.      ∇REPEAT X
      [1]  X, ' ', X, ' ', X, ' ', X, ' ', X
      [2]  ∇

```

All that *REPEAT* does is take the character data you enter (such as 'ANN'), catenate it to a single blank (' '), take it again, catenate it to another blank, and so on.

REPEAT 'ANN'  
ANN ANN ANN ANN ANN

$\forall$ SQUARE X  
[1] 'THE NUMBER SQUARED IS ',  $\forall X \times X$   
[2]  $\forall$   
SQUARE 247  
THE NUMBER SQUARED IS 61009

3.  $\forall A$  CHOOSE B  
[1] A[B]  
[2]  $\forall$   
NOS CHOOSE 4  
21

4.  $\forall X$  OF Y  
[1]  $X \rho Y$   
[2]  $\forall$   
4 OF '□'  
□□□□

5.  $\forall HIST$  D  
[1] D[1] OF '□'  
[2] D[2] OF '□'  
[3] D[3] OF '□'  
[4] D[4] OF '□'  
[5]  $\forall$   
HIST 4 1 6 7  
□□□□  
□  
□□□□□□  
□□□□□□□□

E.

1.  $\forall R \leftarrow A$  GT B  
[1]  $R \leftarrow (A > B) / A$   
[2]  $\forall$   
SALES GT 200  
350 201 279

---

```

2.      ∇R←A LT B
      [1]  R←(A<B)/A
      [2]  ∇
          SALES LT 200
125 115

```

```

3.      ∇R←A EQ B
      [1]  R←(A=B)/B
      [2]  ∇
          SALES EQ 350
350

```

F.

```

      ∇DEPOSIT AMOUNT
[1]  DEPOS←DEPOS,AMOUNT
[2]  ∇
      ∇CHECK AMOUNT
[1]  WITHD←WITHD,AMOUNT
[2]  ∇
      ∇NEWBAL
[1]  (+/DEPOS)-+/WITHD
[2]  ∇

```

You may have forgotten the parentheses around `+/DEPOS` in `NEWBAL`. The function works this way (that is, you don't get any error messages), but you won't get the result you wanted. Instead, the total withdrawals are subtracted from *every* deposit (`DEPOS-+/WITHD`); then those results are summed (`+/`).

## Chapter 6 A.

```

1.      'I8' □FMT D
      10251   17513   18016
      11206   16867   19197

```

Always remember to put quotes around the format phrases; otherwise you'll get a value error.

```

2.      'F10.2' □FMT D
      10250.50 17512.70 18016.20
      11205.81 16867.12 19197.20

```

3. 'F8.2' □FMT D  
 10250.5017512.7018016.20  
 11205.8116867.1219197.20

There was enough room to format the numbers (we didn't get any stars), but we didn't allow for any space between the numbers.

4. 'F18.2' □FMT D  
 10250.50                      17512.70                      18016.20  
 11205.81                      16867.12                      19197.20

5. 'CF18.2' □FMT D  
 10,250.50                      17,512.70                      18,016.20  
 11,205.81                      16,867.12                      19,197.20

6. 'CP<\$>F18.2' □FMT D  
 \$10,250.50                      \$17,512.70                      \$18,016.20  
 \$11,205.81                      \$16,867.12                      \$19,197.20

The order of the phrases *C* and *P* makes no difference.

- B. 1. 'F12.2,F10.1' □FMT E  
 18147.70                      20.4  
 19717.11                      -17.3

Always remember to separate format phrases with a comma, or you'll get a *FORMAT ERROR*.

2. 'CP<\$>F12.2,F10.1' □FMT E  
 \$18,147.70                      20.4  
 \$19,717.11                      -17.3

3. 'CP<\$>F12.2,N<°/°>Q<°/°>F10.1' □FMT E  
 \$18,147.70                      20.4°/°  
 \$19,717.11                      -17.3°/°

If you only used *N<°/°>*, then you only got a percentage sign next to the negative number. If you only used *Q<°/°>*, you only got one next to the positive number. Again, the *order* of the format phrases is not important.

4. 'CP<\$>F12.2,M<->N<°/°>Q<°/°>F10.1' □FMT E  
 \$18,147.70                      20.4°/°  
 \$19,717.11                      -17.3°/°

5. `'CP<$>F12.2,M<->N<./.>P<+>Q<./.>F10.1' □FMT E`  

|             |          |
|-------------|----------|
| \$18,147.70 | +20.4./. |
| \$19,717.11 | -17.3./. |
6. `'8A1,CP<$>F12.2,M<->N<./.>P<+>Q<./.>F10.1' □FMT (N;E)`  

|          |             |          |
|----------|-------------|----------|
| REVENUES | \$18,147.70 | +20.4./. |
| EXPENSES | \$19,717.11 | -17.3./. |

If you forgot the repetition factor for the names (8A1), you got stars in your result.

C.

1. `'F12.2' □FMT S`  

|         |         |
|---------|---------|
| 6543.10 | 5342.55 |
| 6231.89 | 7531.21 |
2. `'F12.2' □FMT (S;T)`  

|         |         |          |
|---------|---------|----------|
| 6543.10 | 5342.55 | 11885.65 |
| 6231.89 | 7531.21 | 13763.10 |
3. `'P<$>F12.2' □FMT (S;T)`  

|           |           |            |
|-----------|-----------|------------|
| \$6543.10 | \$5342.55 | \$11885.65 |
| \$6231.89 | \$7531.21 | \$13763.10 |
4. `'CP<$>F12.2' □FMT (S;T)`  

|            |            |             |
|------------|------------|-------------|
| \$6,543.10 | \$5,342.55 | \$11,885.65 |
| \$6,231.89 | \$7,531.21 | \$13,763.10 |
5. `'9A1,3CP<$>F12.2' □FMT (N1;S;T)`  

|           |            |            |             |
|-----------|------------|------------|-------------|
| SALESREP1 | \$6,543.10 | \$5,342.55 | \$11,885.65 |
| SALESREP2 | \$6,231.89 | \$7,531.21 | \$13,763.10 |

If you forgot the repetition factor for the numbers (a 3 before CP) you got stars for the second two numbers. This is because `□FMT` tried to format the numbers with A, which isn't allowed. If you omitted the repetition factor for the names (9A1), your names turned out very strangely—with long strings of stars.

---

## D.

1.        `▽REPT DATA`  
       [1]    `TOT←+/DATA`  
       [2]    `'9A1,3CP<$>F12.2' □FMT (N1;DATA;TOT)`  
       [3]    `▽`  
           `REPT S`  

|                        |                         |                         |                          |
|------------------------|-------------------------|-------------------------|--------------------------|
| <code>SALESREP1</code> | <code>\$6,543.10</code> | <code>\$5,342.55</code> | <code>\$11,885.65</code> |
| <code>SALESREP2</code> | <code>\$6,231.89</code> | <code>\$7,531.21</code> | <code>\$13,763.10</code> |
  
2.        `▽RPT2 DATA`  
       [1]    `'`  
               `SALES FIGURES'`  
       [2]    `'`  
                   `WEEK1`            `WEEK2`            `TOTALS'`  
       [3]    `TOT←+/DATA`  
       [4]    `'9A1,3CP<$>F12.2' □FMT (N1;DATA;TOT)`  
       [5]    `▽`  
           `RPT2 S`  
               `SALES FIGURES`  
                   `WEEK1`            `WEEK2`            `TOTALS`  

|                        |                         |                         |                          |
|------------------------|-------------------------|-------------------------|--------------------------|
| <code>SALESREP1</code> | <code>\$6,543.10</code> | <code>\$5,342.55</code> | <code>\$11,885.65</code> |
| <code>SALESREP2</code> | <code>\$6,231.89</code> | <code>\$7,531.21</code> | <code>\$13,763.10</code> |

## Chapter 8 A.

1.        `'EXERCISE' □FCREATE 6`  
       (You may have used a different tie number. If so, your tie number will differ in the rest of the answers to Exercise A.)
  
2.        `MSG←'FILE FOR EXERCISES IN CHAPTER 8.'`  
           `MSG □FAPPEND 6`  
       Don't forget the quotes around the message.
  
3.        `DATA←3 3ρ110 150 117 35 65 79 84 114 97`  
           `DATA □FAPPEND 6`
  
4.        `DATA2←11.52 3.72 6.15`  
           `DATA2 □FAPPEND 6`
  
5.        `ITEMS←□FREAD 6 2`
  
6.        `PRICES←□FREAD 6 3`

7.        *PRICES*[3]  
6.15

8.        *ITEMS*[3;]  
84 114 97

9.        *+/ITEMS*  
377 179 295

10.       *+/ITEMS*[3;]  
295

11.       *ITEMS*[3;]×*PRICES*[3]  
516.6 701.1 596.55

This expression gives you the total sales for *each* week. To find the total sales for all three weeks (the grand total):

*+/ITEMS*[3;]×*PRICES*[3]  
1814.25

or

*PRICES*[3]×*+/ITEMS*[3;]  
1814.25

**B.**

1.        *⊞FUNTIE* 6 (or whatever tie number you were using)

2.        *▽TIE*  
[1]      '*EXERCISE*' *⊞FTIE* 277  
[2]      *▽*

3.        *TIE*  
          *⊞FNAMES*  
          *EXERCISE*  
          *⊞FNUMS*  
277

(If you have more than one file currently tied, the display may be different.)



```
4.      ▽READ
      [1]  □FREAD 277 2
      [2]  ▽
           READ
      110 150 117
      25  65  79
      84 114  97

5.      ▽TOTSALE
      [1]  ITEMS←□FREAD 277 2
      [2]  PRICES←□FREAD 277 3
      [3]  PRICES×+/ITEMS
      [4]  ▽
           TOTSALE
      4343.04 655.88 1814.25
```

C.

```
1.      ▽READ DATA
      [1]  □FREAD 277,DATA
      [2]  ▽
           ITEMS←2
           PRICES←3
           READ PRICES
      11.52 3.72 6.15
```

Don't forget the comma between 277 and DATA in your function.

```
2.      ▽TOTAL DATA
      [1]  MAT←□FREAD 277,DATA
      [2]  +/MAT
      [3]  ▽
           TOTAL ITEMS
      377 179 295
```

```
3.      ∇TOTSALE ITEM
      [1]  IT←∅FREAD 277 2
      [2]  PR←∅FREAD 277 3
      [3]  IT[ITEM;]×PR[ITEM]
      [4]  ∇
          NUTS←1
          BOLTS←2
          WRENCHES←3
          TOTSALE NUTS
1267.2 1728 1347.84
          TOTSALE BOLTS
130.2 241.8 293.88
```

Notice that we avoided calling the data we read in on lines [1] and [2] of the function *ITEMS* or *PRICES*. Remember, we assigned the number 2 to *ITEMS* and 3 to *PRICES* in Exercise C.1. We might want to use those variables again, so we don't want to confuse things by reassigning them now. To be really safe, you may want to assign the key variables inside the function.

```
      ∇ TOTSALE ITEM
      [1]  NUTS←1
      [2]  BOLTS←2
      [3]  WRENCHES←3
      [4]  IT←∅FREAD 277 2
      [5]  PR←∅FREAD 277 3
      [6]  IT[ITEM;]×PR[ITEM]
      [7]  ∇
```

This will ensure that the computer will find what it needs when it looks for a value in *NUTS* or *BOLTS*. It takes up more computer time to run the function, however.

D.

1. 'SALES' ∅FCREATE 9  
(You can use any whole number that you're not already using.)
2. SNOS ∅FAPPEND 9  
 MON ∅FAPPEND 9  
 SALES ∅FAPPEND 9

3.        `∇FUNTIE 9`
4.        `∇GETDATA`  
      `[1]    'SALES' ∇FTIE 1076`  
      `[2]    SNOS←∇FREAD 1076 1`  
      `[3]    MON←∇FREAD 1076 2`  
      `[4]    SALES←∇READ 1076 3`  
      `[5]    ∇`
5. (Enter function as shown.)
6.        `∇MONTH NUM`  
      `[1]    M←NUM`  
      `[2]    ∇`
7.        `∇PRINT`  
      `[1]    (MON=M)/(SNOS=S)/[1]SALES`  
      `[2]    ∇`

## Chapter 9

You may use different variable and argument names in the function headers than those shown here. Also, remember that there are *many* ways to work a problem in APL. The solutions shown here are the ones we feel are the simplest and most direct, but they are certainly not the *only* solutions.

- A.
1.        `∇ ASKDIE`  
      `[1]    L1: ?6`  
      `[2]    'TRY AGAIN? (YES OR NO):'`  
      `[3]    →('Y'=1↑∇)ρL1`  
      `[4]    ∇`
  2.        `∇ ADDER;NOS`  
      `[1]    'ENTER NUMBERS TO BE ADDED:'`  
      `[2]    NOS←∇`  
      `[3]    'THE TOTAL IS ',(∇+/NOS),'. '`  
      `[4]    ∇`
-

You may not have used a local variable, but it's a good idea to start using this tool.

```
3.      ▽ ASKREPEAT;THING;X
      [1]  'ENTER THING TO BE REPEATED:'
      [2]  THING←□
      [3]  'REPEAT HOW MANY TIMES?'
      [4]  X←□
      [5]  XρTHING
      [6]  ▽

4.      ▽ ASKHIST;NUMS
      [1]  'ENTER FOUR NUMBERS FOR BAR CHART:'
      [2]  NUMS←□
      [3]  HIST NUMS
      [4]  ▽
```

B.

```
      ▽ GETNAME
      [1]  NAME←□FREAD 1068 2
      [2]  'NAME OF ITEM:'
      [3]  NEWNAME←20↑□
      [4]  NAME←NAME,[1]NEWNAME
      [5]  NAME □FREPLACE 1068 2
      [6]  ▽

      ▽ GETDETAILS
      [1]  DETAILS←□FREAD 1068 3
      [2]  'SERIAL NUMBER, DATE PURCHASED, PRICE:'
      [3]  NEWD←35↑□
      [4]  DETAILS←DETAILS,[1]NEWD
      [5]  DETAILS □FREPLACE 1068 3
      [6]  ▽

      ▽ INVSTART
      [1]  'INVENTOR' □FTIE 1068
      [2]  ▽
```

C.

```
▽ BUDGET
[1] 'ENTER BUDGET ITEM ("END" TO STOP):'
[2] INAMES←20↑□
[3] INAMES←1 20ρINAMES
[4] LP: 'NEXT ITEM:'
[5] NEWLINE←20↑□
[6] →(∧/'END'=3↑NEWLINE)ρNUMS
[7] NEWLINE←1 20ρNEWLINE
[8] INAMES←INAMES,[1]NEWLINE
[9] →LP
[10] NUMS:'ENTER BUDGET FIGURES FOR ',(↑1↑ρINAMES),'
ITEMS:'
[11] N←□
[12] TOT←+ /N
[13] PERCENT←100×N+TOT
[14] ' '
[15] 'BUDGET IS:'
[16] ' '
[17] FSB←'20A1,P<$>CF12.2,N<•/•>Q<•/•>F12.1'
[18] FSB □FMT (INAMES;N;PERCENT)
[19] 44 ρ '-'
[20] T←20↑'TOTAL'
[21] T←1 20ρT
[22] FSB □FMT (T;+ /N;100)
[23] 'DON'T FORGET TO )SAVE IF THIS IS THE FINAL
VERSION.'
[24] ▽
```

Notice that we intentionally didn't make the variables local ones because *SHOWBUDGET* needs the variables to run.

## Appendix B

### Error Messages and What to Do about Them

---

In using this book, and typing in APL expressions, you may have gotten at least a few error messages along the way. It's easy to make errors—all you have to do is misspell a variable or function name or forget temporarily how a function is used.

Luckily though, nothing is really hurt when an error message is displayed. With APL, you can always try again by reexecuting the line (with your corrections).

This appendix can help you make those corrections so you can get started again. Of course, some errors (like *SYNTAX ERROR*) have many possible causes, so it may be difficult to identify all of them. But we can at least help you determine the problem. The following table lists some of the common APL error messages; your APL system may have more than those listed here. Refer to your system documentation for a complete list.

| Error             | Probable Cause                                                                                                                                                                                                                                                                                                                                            | Solution                                                                                                                                                                                                                                |
|-------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>DEFN ERROR</i> | This is a “definition error,” a category for errors that can occur when trying to create user-defined functions. Some common causes are: <ol style="list-style-type: none"><li>1. Trying to create a function with the same name as an existing function.</li><li>2. Making an error using the del editor commands, such as<br/>[ ]<br/>[ ' ' ]</li></ol> | <ol style="list-style-type: none"><li>1. Check <i>)FNS</i> and <i>)VARS</i> to see whether the name already exists. Erase the old function or use another function name.</li><li>2. Check your typing of the editor commands.</li></ol> |

## DOMAIN ERROR

3. Trying to display a function that does not exist, such as:  
 $\nabla \text{NOGOOD}[]$

3. If your spelling is correct, check  $\nabla \text{FNS}$  and  $\nabla \text{VARs}$  to make sure the function exists.

1. You tried to use a function with data it is not intended to work with—character data with a function expecting numeric data, for example.

In this case, the solution is simply not to misuse the function. All you can do is recognize that you are using the function incorrectly (and why), and try again using the correct data type. The function may require your data to be reshaped; for example, you may need to ravel a matrix into a vector.

### Examples

$2 + 'JOHN'$

(uses character data  $'JOHN'$ , when numeric data was expected)

$4 \wedge 0$

(uses numeric data, 4, where logical data is expected)

$'FILE' \square \text{FCREATE}^{-1}$

(uses a negative number,  $-1$ , where a positive number was expected)

2. The answer is not defined.

### Examples

$4 \div 0$  (division by zero)

$^{-1} * .5$  (square root of  $^{-1}$ )

## FILE ARGUMENT ERROR

1. You have used too many letters in the file's name.

1. Shorten the name.

2. You have used illegal characters in the file name, or the first character of your file name is a number.

2. Avoid using the illegal characters.

|                                |                                                                                                                                                                                                       |                                                                                                                                                                                                                          |
|--------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>FILE FULL</i>               | Any more additions to the file would make it exceed its reserved storage maximum.                                                                                                                     | Use a different file, eliminate the file, or eliminate unnecessary components in the present file.                                                                                                                       |
| <i>FILE INDEX ERROR</i>        | You have tried to read beyond the end of the file (such as trying to read the 40th component in a 39-component file) or before the beginning of the file.                                             | Use <code>□FSIZE</code> to find out how many components there are, and do not read beyond the last one or before the first one.                                                                                          |
| <i>FILE NAME ERROR</i>         | You have attempted to create a file with the <i>same</i> name as another file within the same library.                                                                                                | Use <code>□FLIB</code> to look at your list of files, and choose a new name or erase the old file.                                                                                                                       |
| <i>FILE NOT FOUND</i>          | The file you are looking for does not exist in the specified library or on the specified disk.                                                                                                        | Check the spelling of the file name.<br>Also, use <code>□FLIB</code> to check whether you are looking in the right place. (On a personal computer, try another disk.)                                                    |
| <i>FILE TIE ERROR</i>          | <ol style="list-style-type: none"><li>1. You are trying to use a tie number that has no file tied to it.</li><li>2. You are trying to tie a file to a file number that is currently in use.</li></ol> | <ol style="list-style-type: none"><li>1. Tie the file and try again.</li><li>2. Check <code>□FNUMS</code> to see what file tie numbers are already in use. Choose another tie number, or untie the other file.</li></ol> |
| <i>FILE TIE QUOTA EXCEEDED</i> | You have already tied the maximum number of files and can tie no more.                                                                                                                                | Use <code>□FUNTIE</code> to untie some of the unneeded files.                                                                                                                                                            |



***INCORRECT  
COMMAND***

You have used a system command [like `)ERASE` or `)COPY`] incorrectly or without all the necessary parts.

You may have made a typing error. If not, study the command and make sure you have included all necessary information.

***INDEX ERROR***

You have used an invalid index (“subscript”) with a variable. For example, you may have asked for the 11th element of a 10-element vector, or asked for the 2.5 element or the  $-2$  element. This error also occurs for matrices. For example, you may have asked for the element in the second row, third column in a  $2 \times 2$  matrix.

This error often occurs when you go beyond the final element of a variable. Check the shape of the variable (`ρVARIABLE`) and don’t index beyond it.

***LENGTH ERROR***

You are attempting an operation with two variables whose shapes don’t match, such as  
`1 2 3 + 4 5 6 7`.  
Also, some functions are defined for use only on scalars, vectors, or matrices. For example, `1 4 5 6` produces a *LENGTH ERROR*.

Adjust the shapes of the variables if necessary, using `reshape (ρ)`, `take (↑)`, `drop (↓)`, or `ravel (,)`.

***LIMIT ERROR***

You have entered a number that is too large or too small for the system to handle. Or, the rank of your variable is too large for the system, for example:

`15 * 265`

Use some other means of representing the number or the variable—such as scaling it by 1000, or dividing it into parts.

|                          |                                                                                                                                                                                                    |                                                                                                                                                                                                                                                                                                                                                                                |
|--------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>NONCE ERROR</i>       | You have used a feature that is present on some, but not all, APL systems. This is an “error for the nonce.”                                                                                       | Don’t use the feature.                                                                                                                                                                                                                                                                                                                                                         |
| <i>RANK ERROR</i>        | You have used a function with data of incorrect rank—that is, an incorrect number of dimensions. A common cause of this is trying to index a matrix without using a semicolon inside the brackets. | Check the shape of your data, and make sure the function you’re using can accept data of that shape. When indexing matrices, remember to include a semicolon between the row and column indexes (such as <i>MAT</i> [ 2 ; 3 ] ).                                                                                                                                               |
| <i>STACK FULL</i>        | This is most often caused by a user-defined function executing an endless loop or calling a function recursively.                                                                                  | Check your user-defined function for problems or reduce number of recursive calls.                                                                                                                                                                                                                                                                                             |
| <i>SYMBOL TABLE FULL</i> | You have used too many names of user-defined functions and variables, also called “symbols.” APL has no more space for names.                                                                      | Use <i>)SYMB</i> to determine the allowable number of symbols in the symbol table. You must reset the symbol table in a clear workspace. Enter the following sequence of commands to set the maximum number of symbols to 1024.<br><br><i>)SIC</i><br><i>)SAVE TEMP</i><br><i>)CLEAR</i><br><i>SYMBOLS 1024</i><br><i>)COPY TEMP</i><br><i>)SAVE MYWS</i><br><i>)DROP TEMP</i> |

**SYNTAX ERROR**

Your APL expression does not conform to APL syntax (rules of expression). This can be for any of the following reasons:

Most of these causes point directly to their solution: check your typing.

1. You haven't used ( ) or [ ] in matched pairs.
2. You're using the wrong number of arguments—for example, you've tried to use a monadic function dyadically.
3. Two variables are not separated by a function (such as typing `VAR1 VAR2` when you meant to type `VAR1 + VAR2`).

1. Insert an extra ( , ), [, or ].
2. Check the function to see whether it's monadic, niladic, or dyadic.
3. Insert a function between the two variable names.

**VALUE ERROR**

1. You have typed the name of a variable or function in the active workspace that does not exist.
2. You have misspelled a user-defined function or variable name.
3. You have attempted to use the result of a user-defined function that was not defined to return an explicit result.

1. Check your typing.
2. Check the `)FNS` or `)VARS` to see whether the user-defined function or variable exists.
3. Change the definition of the user-defined function so that it returns an explicit result.

***WS FULL***

You have attempted a problem that created a result (or temporary result, invisible to you) that needed more space than is available in the the active workspace.

Try erasing some unneeded variables or user-defined functions to free up some space. Localize variables within user-defined functions. Place variables or functions in files and read and write to files.

***WS NOT FOUND***

The workspace you are looking for does not exist in the specified or default library or on the specified or default disk.

Check the spelling of the workspace name. Also, use `⎕WSLIB` to check whether you are looking in the right library. On personal computers, try another disk.

## Appendix C

### Keyboard Template

---

Use this blank keyboard as a guide to the location of the APL symbols discussed in this book. Find the location of the symbols in your APL system on your specific keyboard; then write them in the appropriate location below. This template shows the keys most common to all keyboards; the letters and numbers are shown on the keys.

|          |   |   |   |   |   |   |   |   |   |   |   |
|----------|---|---|---|---|---|---|---|---|---|---|---|
| 1        | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0 | — | = |
| Q        | W | E | R | T | Y | U | I | O | P | [ | ] |
| A        | S | D | F | G | H | J | K | L | ; | ' |   |
| Z        | X | C | V | B | N | M | , | . | / |   |   |
| Spacebar |   |   |   |   |   |   |   |   |   |   |   |

## Appendix D

### APL Programs Used in This Book

---

```
▼ACCOUNT V
[1]  I/V
[2]  I/V
[3]  +\V
[4]  +/V
▼

▼ACCOUNT2 DATA
[1]  'THE LARGEST DEPOSIT IS ',⌈I/ DATA
[2]  'THE LARGEST WITHDRAWAL IS ',⌈I/ DATA
[3]  'THE RUNNING BALANCE IS ',⌈+\ DATA
▼

▼R←A ADD B
[1]  R←A+B
▼

▼ADDER;NOS
[1]  'ENTER NUMBERS TO BE ADDED'
[2]  NOS←□
[3]  'THE TOTAL IS ',(⌈+\ NOS),'. '
▼

▼ADDREC
[1]  'ENTER YOUR RECIPE LINE BY LINE (USE "END" TO
STOP):'
[2]  RECIPE←60↑□
[3]  RECIPE← 1 60 ρRECIPE
[4]  L1:'NEXT LINE:'
[5]  NEWLINE←60↑□
[6]  →(∧/'END'=3↑NEWLINE)ρL2
[7]  NEWLINE← 1 60 ρNEWLINE
[8]  RECIPE←RECIPE,[1]NEWLINE
[9]  →L1
[10] L2:RECIPE ⌈FAPPEND 1111
[11] DIR←⌈FREAD 2222 1
[12] NEWENTRY←RECIPE[1;]
[13] DIR←DIR,[1]NEWENTRY
[14] DIR ⌈FREPLACE 2222 1
[15] 'NEW RECIPE FILED.'
▼
```

```
▼ASKAVERAGE
[1]  'WHAT NUMBERS DO YOU WANT TO AVERAGE?'
[2]  VECTOR←□
[3]  ANSWER←(+÷VECTOR)÷ρVECTOR
[4]  'THE AVERAGE IS ',(⌘ANSWER),'. '
```

▼

```
▼ASKDICE
[1]  +/? 6 6
[2]  'TRY AGAIN? (YES OR NO):'
[3]  →('Y'=1↑□)÷1
```

▼

```
▼ASKDICE2
[1]  LINE1: +/? 6 6
[2]  'TRY AGAIN? (YES OR NO):'
[3]  →('Y'=1↑□)ρLINE1
```

▼

```
▼ASKDICE2
[1]  LINE1: +/? 6 6
[2]  'TRY AGAIN? (YES OR NO):'
[3]  →('Y'=1↑□)ρLINE1
```

▼

```
▼ASKDIE
[1]  L1: ?6
[2]  'TRY AGAIN? (YES OR NO):'
[3]  →('Y'=1↑□)ρL1
```

▼

```
▼ASKDIE2
[1]  LINE1: ?6
[2]  'TRY AGAIN? (YES OR NO):'
[3]  →('Y'=1↑□)ρLINE1
```

▼

```
▼ASKHIST;NUMS
[1]  'ENTER FOUR NUMBERS FOR BAR CHART'
[2]  NUMS←□
[3]  HIST NUMS
```

▼

```

      ▼ASKREPEAT,THING;X
[1]  'ENTER THING TO BE REPEATED:'
[2]  THING←□
[3]  'REPEAT HOW MANY TIMES?'
[4]  X←□
[5]  XρTHING
      ▼

      ▼AVG DATA
[1]  (+÷DATA)÷ρDATA
      ▼

      ▼R←AVG2 A
[1]  R←(+÷A)÷ρA
      ▼

      ▼BUDGET
[1]  'ENTER BUDGET ITEM ("END" TO STOP):'
[2]  INAMES←20↑□
[3]  INAMES← 1 20 ρINAMES
[4]  LP: 'NEXT ITEM:'
[5]  NEWLINE←20↑□
[6]  →(∧/'END'=3↑NEWLINE)ρNUMS
[7]  NEWLINE← 1 20 ρNEWLINE
[8]  INAMES←INAMES,[1]NEWLINE
[9]  →LP
[10] NUMS: 'ENTER BUDGET FIGURES FOR ',(⊖1↑ρINAMES),'
ITEMS'
[11] N←□
[12] TOT←+÷N
[13] PERCENT←100×N÷TOT
[14] ''
[15] 'BUDGET IS:'
[16] ''
[17] FSB←'20A1,P<$>CF12.2,N<0/0>Q<0/0>F12.1'
[18] FSB ρFMT(INAMES;N;PERCENT)
[19] 44ρ '-'
[20] T←20↑'TOTAL'
[21] T← 1 20 ρT
[22] FSB ρFMT(T;+÷N;100)
[23] 'DON'T FORGET TO )SAVE IF THIS IS THE FINAL
VERSION.'
      ▼

```

---



```
    ▽BUILDDIRECTORY
[1]  DIR←DFREAD 2222 1
[2]  'ENTER TITLE OF RECIPE:'
[3]  NEWENTRY←60↑␣
[4]  NEWENTRY← 1 60 ρNEWENTRY
[5]  DIR←DIR,[1]NEWENTRY
[6]  DIR ♂FREPLACE 2222 1
    ▽

    ▽CALC
[1]  INCOMEDATA[3;]←INCOMEDATA[1;]-INCOMEDATA[2;]
    ▽

    ▽CHECK BUCKS
[1]  WITHD←WITHD,BUCKS
    ▽

    ▽A CHOOSE B
[1]  A[B]
    ▽

    ▽COLTOT MAT
[1]  +/[1]MAT
    ▽

    ▽COUNT
[1]  ONE
[2]  TWO
[3]  TOO
[4]  TREE
[5]  FORE
    ▽

    ▽DEPOSIT BUCKS
[1]  DEPOS←DEPOS,BUCKS
    ▽

    ▽DIE
[1]  ?6
    ▽

    ▽ANS←DIE2
[1]  ANS←?6
    ▽
```

---

```

      ▼ENTER
[1]  GETNAME
[2]  GETDETAILS
[3]  'NEW ITEM FILED.'
      ▼

      ▼R←A EQ B
[1]  R←(A=B)∕B
      ▼

      ▼ERASEBLANK S
[1]  (' '=S)∕S
      ▼

      ▼FILEREcipe
[1]  'RECIPES' OFTIE 9971
[2]  'ENTER YOUR RECIPE LINE BY LINE (USE "END" TO
STOP):'
[3]  RECIPE←60↑□
[4]  RECIPE← 1 60 ρRECIPE
[5]  L1:'NEXT LINE:'
[6]  NEWLINE←60↑□
[7]  →(∧/'END'=3↑NEWLINE)ρL2
[8]  NEWLINE← 1 60 ρNEWLINE
[9]  RECIPE←RECIPE,[1]NEWLINE
[10] →L1
[11] L2:RECIPE OFAPPEND 9971
[12] OFUNTIE 9971
[13] 'DONE.  NEW RECIPE FILED.'
      ▼

      ▼FILESales;SALES
[1]  A PUT NEW SALES FIGURES IN SALES FILE
[2]  'SALESFIG' OFTIE 2201
[3]  'ENTER SALES FIGURES FOR THIS WEEK'
[4]  SALES←□
[5]  SALES OFAPPEND 2201
[6]  OFUNTIE 2201
[7]  'NEW SALES FILED.'
      ▼
```

```
▼FLIP
[1] N← 2 5p'HEADTAILS'
[2] ((?2)= 1 2)/[1]N
[3] 'TRY AGAIN? (YES OR NO):'
[4] →('Y'=1↑□)p2
▼

▼FLIP2
[1] N← 2 5p'HEADTAILS'
[2] MORE:((?2)= 1 2)/[1]N
[3] 'TRY AGAIN? (YES OR NO):'
[4] →('Y'=1↑□)pMORE
▼

▼GETDATA
[1] 'SALES' □FTIE 1076
[2] SNOS←□FREAD 1076 1
[3] MON←□FREAD 1076 2
[4] SALES←□FREAD 1076 3
▼

▼GETDETAILS
[1] DETAILS←□FREAD 1068 3
[2] 'SERIAL NUMBER, DATE PURCHASED, AND PRICE:'
[3] NEWD←35↑□
[4] DETAILS←DETAILS,[1]NEWD
[5] DETAILS □FREPLACE 1068 3
▼

▼GETNAME
[1] NAME←□FREAD 1068 2
[2] 'NAME OF ITEM:'
[3] NEWNAME←20↑□
[4] NAME←NAME,[1]NEWNAME
[5] NAME □FREPLACE 1068 2
▼

▼R←A GT B
[1] R←(A>B)/A
▼
```

---

▼GUESS

```
[1] 'I'M THINKING OF A NUMBER BETWEEN 1 AND 3.'
[2] 'WHAT IS IT?'
[3] LOOP:'ENTER YOUR NUMBER.'
[4] NUM←□
[5] ANS←?3
[6] →(ANS=NUM)ρOK
[7] 'I'M SORRY. THE NUMBER WAS ',(⌘ANS),'. '
[8] 'TRY AGAIN?'
[9] →('Y'=1↑□)ρLOOP
[10] →0
[11] OK:'RIGHT! TRY AGAIN?'
[12] →('Y'=1↑□)ρLOOP
```

▼

▼GUESS2

```
[1] 'I'M THINKING OF A NUMBER BETWEEN 1 AND 3.'
[2] 'WHAT IS IT?'
[3] LOOP:'ENTER YOUR NUMBER'
[4] NUM←□
[5] ANS←?3
[6] →(ANS=NUM)ρOK
[7] 'I'M SORRY, THE NUMBER WAS ',(⌘ANS),'. '
[8] 'TRY AGAIN?'
[9] →('Y'=1↑□)ρLOOP
[10] →0
[11] OK:'RIGHT! TRY AGAIN?'
[12] →('Y'=1↑□)ρLOOP
```

▼

▼R←HALF NUM

```
[1] R←NUM×0.5
```

▼

▼HELLO

```
[1] 'MY NAME IS JERRY. WHAT'S YOURS?'
[2] A←□
[3] 'NICE TO MEET YOU, ',A,'. '
```

▼

▼HIST D

```
[1] D[1] OF '□'
[2] D[2] OF '□'
[3] D[3] OF '□'
[4] D[4] OF '□'
```

▼

```

      ▼INITBAL AMOUNT
[1]  DEPOS←AMOUNT
[2]  WITHD←0
      ▼

      ▼INVSTART
[1]  'INVENTOR' ⌈FTIE 1068
      ▼

      ▼LIST;DIR;NUMS;NUMBER;NAMES;NAM;DET
[1]  ⌈ MANAGES HOUSEHOLD INVENTORY SYSTEM
[2]  L1:'LIST, ITEMS, OR QUIT? (D,I,Q):'
[3]  →('DIQ'=1↑□)/DI,ITEMS,EXIT
[4]  'PLEASE ENTER D, I, OR Q'
[5]  →L1
[6]  DI:DIR←⌈FREAD 1068 2
[7]  NUMS←1↑↑DIR
[8]  'Q< >I5,20A1' ⌈FMT(NUMS;DIR)
[9]  →L1
[10] ITEMS:'LIST WHICH ITEM (BY NUMBER)?'
[11] NUMBER←□
[12] NAMES←⌈FREAD 1068 2
[13] DETAILS←⌈FREAD 1068 3
[14] NAM←NAMES[NUMBER;]
[15] DET←DETAILS[NUMBER;]
[16] NAM,DET
[17] →L1
[18] EXIT:
      ▼

      ▼LISTDIRECTORY
[1]  DIR
      ▼

      ▼R←A LT B
[1]  R←(A<B)/A
      ▼

      ▼MONTH NUM
[1]  M←NUM
      ▼

      ▼NEWBAL
[1]  (+/DEPOS)-+/WITHD
      ▼
```

```
      ▼NOBLANK CHARVECT
[1]  R REMOVES BLANKS FROM CHARACTER VECTOR
[2]  (CHARVECT≠' ')/CHARVECT
      ▼

      ▼X OF Y
[1]  XρY
      ▼

      ▼R←WIDTH PERIM LENGTH
[1]  R←2×WIDTH+LENGTH
      ▼

      ▼WIDTH PERIMETER LENGTH
[1]  2×WIDTH+LENGTH
      ▼

      ▼PRINT
[1]  (MON=M)/(SNOS=S)/[1]SALES
      ▼

      ▼READ DATA
[1]  ⚡FREAD 277,DATA
      ▼

      ▼READ2
[1]  'RECIPES' ⚡TIE 3371
[2]  'WHAT IS THE NUMBER OF THE RECIPE?'
[3]  ANS←⚡
[4]  ⚡FREAD 3371,ANS
[5]  ⚡FUNTIE 3371
      ▼

      ▼READREC
[1]  'READ WHICH RECIPE (BY NUMBER)'
[2]  RECNO←⚡
[3]  ⚡FREAD 1111,RECNO
      ▼
```

```

      ▼RECIPES
[1]  TIEFILES
[2]  L0:'LIST, READ, AD, OR QUIT? (L, R, A, OR Q):'
[3]  →('LRAQ'=1↑□)/L1,L2,L3,L4
[4]  L1:LISTDIRECTORY
[5]  →L0
[6]  L2:READREC
[7]  →L0
[8]  L3:ADDREC
[9]  →L0
[10] L4:DFUNTIE 1111 2222

```

```

      ▼REPEAT X
[1]  X,' ',X,' ',X,' ',X,' ',X

```

```

      ▼REPORT
[1]  '                                HOMEGROWN CO.'
[2]  ''
[3]  '                                1981            1982            1983'
[4]  ''
[5]  CALC
[6]  '7A1,3CF12.2' DFMT(NAMES;INCOMEDATA)

```

```

      ▼REPT DATA
[1]  TOT←+/DATA
[2]  '9A1,3CP<$>F12.2' DFMT(N1;DATA;TOT)

```

```

      ▼ROWTOT MAT
[1]  +/MAT

```

```

      ▼RPT DATA
[1]  DATA
[2]  '-----'
[3]  COLTOT DATA

```

```

      ▼RPT2 DATA
[1]  '          SALES FIGURES'
[2]  '          WEEK1      WEEK2      TOTALS'
[3]  TOT+÷/DATA
[4]  '9A1,3CP<$>F12.2' ⚡FMT(N1;DATA;TOT)
      ▼

      ▼RUNTOT DATA
[1]  +\DATA
      ▼

      ▼SALESMAN NUM
[1]  S←NUM
      ▼

      ▼SHOWBUDGET
[1]  ⚡ DISPLAYS BUDGET CREATED BY "BUDGET" FUNCTION
[2]  ''
[3]  FSB ⚡FMT(INAMES;N;PERCENT)
[4]  44⚡''
[5]  FSB ⚡FMT(T;÷/N;100)
      ▼

      ▼SQUARE X
[1]  'THE NUMBER SQUARED IS ',⚡X×X
      ▼

      ▼TIE
[1]  'EXERCISE' ⚡TIE 277
      ▼

      ▼TIEFILES
[1]  'RECIPES' ⚡TIE 1111
[2]  'DIRECTRY' ⚡TIE 2222
      ▼

      ▼TOTAL COMPNO
[1]  MAT+⚡FREAD 277,COMPNO
[2]  +/MAT
      ▼
```



```
▼TOTSALE ITEM
[1] IT←OFREAD 277 2
[2] PR←OFREAD 277 3
[3] IT[ITEM; ]×PR[ITEM]
▼
▼TOTSALLES
[1] ITEMS←OFREAD 277 2
[2] PRICES←OFREAD 277 3
[3] PRICES×+/ITEMS
▼
```

## Glossary

---

### **append**

Add data to the end of a file. You normally add new data to a file by appending the data to the file (that is, by using the command `⌘FAPPEND`).

### **character data**

Data that is not used for arithmetic, but which is rather passed back to the user as it is. Character data is distinguished from numeric data by placing single quotes at the beginning and end of the data.

### **character input mode**

State in which the computer accepts character input *without* its being enclosed in single quotes. Used in interactive functions. Also called quote-quad input.

### **column**

The second dimension of a matrix—the vertical arrangement of a matrix's numbers.

### **component**

Single segment of a file. Components are numbered sequentially (1, 2, 3, and so on) and contain only one thing at a time. They can contain either character or numeric data of any size or shape.

### **data**

Numbers, or letters and symbols, that a function does its work on. APL works with data in the form of scalars, vectors, matrices, and even data with more dimensions (which is beyond the scope of this book).

### **DEFN ERROR**

Definition error, results when you try to define a function with too many arguments. Also results when you try to define a function that has the same name as a variable or an existing function in the workspace.

### **del**

The `▽` symbol. Used to identify the beginning and end of a function definition.

**DOMAIN ERROR**

Results when you try to use a function in ways it's not defined to be used, such as  $5 \div 0$  or  $1^{-4}$ .

**dyadic function**

Type of function that has two arguments, or pieces of data. In particular, it has both a left and a right argument. An example is the plus function:  $4 + 5$ . Compare with "monadic function."

**element**

A segment or part of a vector. For example, the third elements of the vectors  $\begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$  and  $\text{'MARY'}$  are 3 and R.

**evaluated input mode**

State in which the system accepts input in the form of numbers or characters enclosed in single quotes. Used in interactive functions. Also called quad input mode.

**explicit result**

A function result that is stored by the system so that it can be used by other functions for further work. Functions *without* explicit results merely display their results; the results are not stored.

**expression**

See "statement."

**file**

Storage place for data that is independent of the workspace. Files must be tied to be active, and they are referred to by their tie numbers.

**FILE INDEX ERROR**

Results when you try to read or replace a file component that doesn't exist.

**FILE NAME ERROR**

Results when the file name is incorrect, usually because of a typing error. Also results if you try to create a file with a name already being used by one of your other files.

---

### **FILE TIE ERROR**

Results if no file is currently tied to the tie number you've entered. Also results if you try to tie a file using a tie number that you're currently using for another file.

### **formatting**

Process of displaying the system's responses (output) in a form different from the one the system usually provides. This includes changing the spacing, inserting commas and other symbols or text, and changing the number of decimal places.

### **function**

List of instructions for performing a task. In APL, functions can be built-in (such as  $+$  and  $\div$ ) or user-defined. Both types of functions use data to do work.

### **function definition mode**

State in which the system accepts the lines of a user-defined function. The system collects these lines and stores them until you run the function. You enter and exit this state by entering a del ( $\nabla$ ).

### **global variables**

Variables that are used throughout the workspace. These variables might be used by a number of functions in the workspace. Compare with "local variables."

### **immediate execution mode**

State in which the computer acts as a desk calculator, executing APL statements and giving answers. The system is always in this state when you sign on.

### **interactive**

Type of function that "interacts" with you by asking you questions and doing its work with the answers.

### **iota**

The  $\iota$  symbol, used by the "count" function.

### **length**

The number of elements in a vector. For example, both of the vectors  $1\ 2\ 3\ 4$  and 'MARY' have a length of four.

**LENGTH ERROR**

Results when you try to use two arguments that don't have the same number of elements, such as `1 2 + 3 4 5`.

**line labels**

Names used to identify lines of a user-defined function. Used together with branching to guide the system's execution of a function. Labels are separated by a colon from APL statements on the same line of the function.

**local variables**

Variables "locked inside" a user-defined function. These variables exist only while a function is being run; they are erased when the program is complete. Compare with "global variables."

**matrix**

A table of data. A matrix has two dimensions—rows and columns. Rows are the first dimension, and columns are the second dimension. Both numeric data and character data can be formed into matrices using the reshape function.

**monadic function**

Type of function with a right argument only. That is, a function that takes data on the right side only. An example is the shape function: `ρVECTOR`. Compare with "dyadic function."

**numeric data**

Numbers used for arithmetic in the computer. Compare with "character data."

**order of execution**

The rule used by APL to solve a problem (that is, to execute a line you enter). APL always begins with the part of the expression farthest to the right, solves it, and then uses the answer to solve the next part of the expression. When APL reaches the left-most part of the expression, it knows that it's finished. APL then either displays the final result or stores the result in a variable, depending on what you've asked it to do.

**quad**

The `□` symbol. Used in function editing commands, in file system functions and other system functions, and to request evaluated input.

**quad input mode**

See “evaluated input mode.”

**quote-quad**

The  $\square$  symbol, used to request character input.

**quote-quad input mode**

See “character input mode.”

**RANK ERROR**

Similar to a *LENGTH ERROR*. Results when you use two arguments that don’t have the correct number of dimensions, such as trying to add a vector (one dimension) to a matrix (two dimensions).

**relational functions**

Built-in APL functions that show a relationship between two numbers. Includes the functions “less than” ( $<$ ), “less than or equal to” ( $\leq$ ), “equals” ( $=$ ), “greater than or equal to” ( $\geq$ ), “greater than” ( $>$ ) and “not equal to” ( $\neq$ ).

**repetition factor**

A number placed at the beginning of a format phrase that indicates how many times the format phrase should be repeated. That is, it shows how many times the system should use the same format phrase to format consecutive columns of data.

**rho**

The  $\rho$  symbol. Used with the functions “shape” and “reshape.”

**row**

The horizontal arrangement of numbers in a matrix. Rows are considered the first dimension of a matrix.

**scalar**

A single number. Compare with “vector” and “matrix.”

**statement**

String of APL symbols and numbers or letters. A statement always contains at least one function (a symbol) and some data (numbers or letters).

**subfunction**

Function called by another function. The main function includes the name of the subfunction as one of its lines; thus the subfunction gets run as part of running the main function.

**syntax**

General form of a function. A function's syntax tells how many arguments a function takes and which side of the function name or symbol they appear on. User-defined functions have three possible syntaxes—one argument (a right argument), two arguments (left and right arguments), or no arguments. Built-in APL functions have only two possible syntaxes—one argument or two arguments.

**SYNTAX ERROR**

Results when you mix up the order of the function and data (such as 2 3 4 + /) or you try to use too many arguments (such as 2 AVG 9 10 11, where AVG has the syntax *AVG data*).

**system commands**

Commands such as *LOAD* and *SAVE* that control the system rather than do arithmetic.

**system functions**

Commands such as *LOAD* that are similar to system commands but may be used inside user-defined functions. Some system functions, such as *FMT*, have capabilities beyond those of system commands.

**tie number**

Whole number associated with a file while the file is active. You must use unique tie numbers for each file you have tied, but you can use different tie numbers for a file each time you tie it.

**VALUE ERROR**

Results when you ask for a variable that doesn't exist. This message often results from a typing error, such as if you entered *DAT* when you meant *DATA*.

**variable**

Storage place for data. Variables have names that can include both letters and numbers, as long as the first character of the name is a letter. Data stored in variables is stored *permanently* when you enter the command `)SAVE`.

**vector**

List, or string, of numbers, each separated by at least one blank space. For example, `7 9 1 1` is a vector.

**workspace**

Place for storing both functions and data. Workspace names must begin with a letter. Workspaces are stored permanently when you enter the command `)SAVE`.



# Index

---

When an item has two or more page numbers noted, the most significant are shown in boldface. Leading APL characters, such as `)` and `⌈`, are ignored in the alphabetical list.

## A

- active workspace 104-106, 108-112, 150
- and function 54
- and reduction 55, 56, 61
- APL 4, 5, 16
- APL idioms 176, 180
- APL symbols 8
- argument 23, 24
- assignment arrow 16
- atomic vector 158
- `⌈AV` 158, 159

## B

- branch function 134-136, 144, 149, 177
- branching 129, 134-136, 144, 177

## C

- calendar 7-9
- catenate function **18-19**
  - with matrices 132
- ceiling 159
- character data **28-32**, 129-131, 155, 158, 163, 177
  - digits as 30
  - in a function 75, 129
  - in programs 64
  - selecting 52
- character input mode 130, 131
- character matrices 31, 44, 57
- character vector 30, 45, 133, 137, 176
- characteristics of files 114
- checking for an empty vector 179
- circular 175
  - `)CLEAR` 11, 43, 69, 90, 104-107, 109-112
- clear workspace 104, 105

- clearing the workspace 11
- combinations function 175
- comment 137, 149
- component of a file 114, 116-118, 121-124, 126
- component number 116-118, 121, 123, 124
  - compression function 51, 52, 56, 58, 60, 62, 131, 132, 144
- conditional branch 134
- `)COPY` 64, 105, 108, 110-112
- correcting errors 15, 68-74
- count function 41-42

## D

- decorations 95
- defining functions 65-90
- DEFN ERROR* 69, 78, 137, 208
- del editor 66
- DEMOAPL* 6, 8, 9, 106
- DESCRIBE* 6, 8, 9
- DOMAIN ERROR* 30, 31, 41, 53, 92, 173, 209
- `)DROP` 109
- drop function 170, 171
- dyadic function 23

## E

- editing a function 67-74
- `)ERASE` 69, 70, 76, 78, 104, 107, 108, 111
- error message 6-8, **208-214**
- evaluated input 130, 131, 148
- evaluated input mode 131
- execute function 162
- explicit result 79-85, 88
- exponentiation function 171

## **F**

`⌘FAPPEND` 115, 116, 121, 123  
`⌘FCREATE` 115, 120, 121  
`⌘FERASE` 120, 121  
`⌘FI` 163  
file 79, **113-124**, 126, 128, 140-153  
    advantages of using 113  
    creating 115  
    defined 113  
    determining size of 116  
    erasing 120  
    listing 120  
    naming 119  
    putting data in 115  
    reading data from 117  
    tying 119  
    untying 118  
    using in functions 140-147  
*FILE ARGUMENT ERROR* 209  
*FILE FULL* 210  
*FILE INDEX ERROR* 117, 147, 210  
*FILE NAME ERROR* 115, 209  
*FILE NOT FOUND* 210  
*FILE TIE ERROR* 118, 119, 121, 210  
*FILE TIE QUOTA EXCEEDED* 210  
`⌘FLIB` 120, 121  
floor function 159  
`⌘FMT` 91, 93-102  
`⌘FNAMES` 119-122  
`⋄FNS` 9, 10, 83, 107, 111  
`⌘FNUMS` 119, 121-123  
format function **31, 91-93**  
format pairs 91, 92  
format phrases 93, 95-97, 100, 102  
format string 96, 97, 99  
formatting 91-103  
`⌘FREAD` 114, 117-119, 121, 123, 124  
`⌘FREPLACE` 118, 121, 123  
`⌘FSIZE` 116, 121  
`⌘FTIE` 119, 121, 123

## **function**

    arguments of 23  
    defined 9, 65  
    dyadic 24, 77  
    editing 68-74  
    explicit result in 79  
    interactive 129  
    listing a definition of 70  
    listing names of 9, 107  
    monadic 24, 76  
    niladic 78  
    syntax of 82  
function definition mode 64, **66-69**, 71-74, 76, 83, 84  
function header 66, 84  
`⌘FUNTIE` 118-121

## **G**

global variables 138, 139, 149, 154  
grade function 157

## **H**

high minus symbol 13, 14

## **I**

immediate execution mode 11, 12, 66, 75, 83, 131  
*INCORRECT COMMAND* 211  
*INDEX ERROR* 40, 211  
index function 28, **37-40**  
indexing 32, **37-40**  
    into a matrix 39  
    together with assignment 38  
    with character data 40  
inner product function 162, 172, 173  
input line editing 15  
interactive function 129, 180  
interrupting input 16

**L**

lamp symbol 137, 149  
□LC 177  
least-squares approximation 175  
LENGTH ERROR 13, 15, 211  
LESSONS 64, 83, 150  
)LIB 104, 106, 108, 109, 111, 112, 180  
library 104, 106, 108, 109, 111, 112, 119-121, 180  
library number 104, 106  
LIMIT ERROR 211  
line labels 135, 136  
)LOAD 43, 104-108, 110-112  
local variables 137, 139, 140, 149  
log function 166, 171  
logarithm function 171

**M**

matrix 19-21, 23, 24, 26  
matrix division function 173  
matrix multiplication function 162, 171, 172  
maximum function 28, 32-35, 37, 44  
maximum reduction 34, 35  
membership function 159  
middle minus symbol 13  
minimum function 28, 34, 35, 37, 44  
minimum reduction 35  
mixing vectors with scalars 13  
monadic function 23  
multidimensional objects 165, 166

**N**

negative number 13, 14  
NONCE ERROR 211  
numeric data 28, 41, 44, 91, 163  
numeric formatting 93

**O**

)OFF 100, 111, 112, 170, 171  
or function 54  
or reduction 56, 61

order of execution 14, 15  
outer product function 171, 172  
overturn function 168, 169

**P**

plus reduction 28, 35, 53, 65, 76, 172  
plus scan 28, 37, 44  
power function 162, 167, 171-173  
program (see function) 4-9, 65-90

**Q**

quad input 129-131, 134, 135, 148, 149  
quote-quad 130, 131, 134, 135, 149  
quote-quad input mode 130

**R**

rank 164-166, 176, 179  
rank error 212  
ravel 166, 167  
reduction 28, 34-37, 53, 55-57, 61, 62, 65, 76, 172  
reduction on a matrix 35  
reduction with multiplication 35  
relational functions 52, 53  
reordering a matrix 158  
repetition factor 97, 98, 100, 102  
replacing an entire line 68  
reshape function 11, 19, 20, 24  
right-to-left execution 14, 24  
roll function 42-44  
rotate function 167-169  
rounding 159  
running total 85

**S**

)SAVE 43, 60, 83, 104-112  
scalar 12, 13, 23  
scan function 28, 37, 44  
selection expression 52, 58

shape function 18-20  
    of a group of characters 29  
    of a matrix 20  
)*SIC* 69, 76, 137  
sorting 158, 169  
*SS* 167, 179  
stack 69  
*STACK FULL* 212  
*SYMBOL TABLE FULL* 212  
syntax 82, 163  
*SYNTAX ERROR* 82, 163, 213  
system command 105-111  
system function 91, 93, 99, 111, 163, 167, 177,  
    179

## T

table lookup function 161, 162  
take function 132  
tie number 114-121, 124  
transpose 169  
trapping errors 145  
trigonometric functions 175

## U

user-defined function 65-90  
using parentheses 15, 24

## V

*VALUE ERROR* 213  
variable 6, 9, 10, 16-18, 22, 26, 27  
    global 138  
    local 138  
vector 12, 13, 19, 20, 23, 24, 26  
    as a "key" to a matrix 58  
)*VAR* 9, 10, 107, 108, 112  
*VI* 163

## W

where function 160  
workspace  
    active 104  
    clearing 11, 105  
    copying 110  
    defined 7, 10, 104  
    dropping 109  
    listing contents of 9, 107  
    listing names of 108  
    loading 7  
    naming 108  
    saving 105  
writing programs 64-90  
*WS FULL* 214  
*WS NOT FOUND* 214  
)*WSID* 108, 109, 112  
)*WSLIB* 108-110, 112